Sun 公司核心技术丛书

JAVA 核心技术 卷 I：基础知识

Core Java , Volume I : Fundamentals Eighth Edition



（美）Cay S. Horstmann
Gary Cornell 著

叶乃文 邝劲筠 杜永萍 译

- 针对Java SE6平台进行了全面更新。
- 涵盖Java语言核心内容。
- 大量精心设计代码示例。
- CSDN Java大版主隆重推荐。



原书第8版



机械工业出版社
China Machine Press

VISIT…

LANZAROTE Caliente.COM

译 者 序

《Java核心技术》自第1版出版以来，一直备受广大Java程序设计人员的青睐，是一本畅销不衰的Java经典书籍。本书的两位作者Cay S.Horstmann和Gary Cornell都具有编写程序设计方面书籍的丰富经验。

众所周知，Java程序设计语言仍处于不断完善和发展的活跃时期，为了能够及时地跟上Java 的前进步伐，在短短的10余年间，本书已经修订了7次，第8版同样是为了适应Java的最新特性而重新修订的。新版主要增加了对Java标准版（Java SE 6）特性的全面介绍，并对第7版中两卷的内容安排做了部分调整。即将第7版第Ⅰ卷中的“流与文件”调到第Ⅱ卷中，将第7版第Ⅱ卷中的“集合”与“多线程”调到第Ⅰ卷中。

我们诚心地向您推荐这本书籍。它囊括了Java 2平台、标准版（J2SE）的全部基础知识。作为一本精练的技术指南和可信赖的参考书籍，其中提供了大量的应用实例，用来说明Java的重要语言规则和库功能，而且，这些示例程序都是完整且具有实际意义的。最重要的是：所有程序都已经被升级为Java SE 6，它们将成为独自编写Java程序的良好开端。

参加本书翻译的有叶乃文、邝劲筠以及杜永萍。书中文字与内容力求忠实原著，但限于译者水平有限，加上时间仓促，译文中难免有疏漏之处，敬请广大读者批评指正。

译者

2008年4月于北京

前 言

致读者

1995年底，Java语言在Internet舞台一亮相便名声大噪。其原因在于它将有望成为连接用户与信息的万能胶，而不论这些信息来自于Web服务器、数据库、信息提供商，还是任何其他渠道。事实上，就发展前景而言，Java的地位是独一无二的。它是一种完全可信赖的程序设计语言，得到了除微软之外的所有厂家的认可。其固有的可靠性与安全性不仅令Java程序员放心，也令使用Java程序的用户放心。Java内建了对网络编程、数据库连接、多线程等高级程序设计任务的支持。

1995年以来，Sun Microsystems公司已经发布了Java开发工具箱（Java Development Kit）的7个主要版本。在过去的11年中，应用程序接口（API）已经从200个类扩展到3000个类，并覆盖了用户界面构建、数据库管理、国际化、安全性以及XML处理等各个不同的领域。

本书是《Java核心技术》第8版的卷Ⅰ。自《Java核心技术》出版以来，每个新版本都尽可能快地跟上Java开发工具箱发展的步伐，而且每一版都重新改写了部分内容，以便适应Java的最新特性。在这一版中，已经反映了Java 标准版（Java SE 6）的特性。

与前几版一样，本版仍然将读者群定位在那些打算将Java应用到实际工程项目中的程序设计人员。本书假设读者是一名具有程序设计语言（除Java之外）坚实背景知识的程序设计人员，并且不希望书中充斥着玩具式的示例（诸如：烤面包机、动物园的动物或神经质地跳动文本）。这些内容绝对不会在本书中出现。本书的目标是让读者充分地理解书中介绍的Java语言及Java类库的相关特性，而不会产生任何误解。

在本书中，我们选用大量的示例代码演示所讨论的每一个语言特性和类库特性。我们有意使用简单的示例程序以突出重点，然而，其中的大部分既不是赝品也没有偷工减料。它们将成为读者自己编写代码的良好开端。

我们假定读者愿意（甚至渴望）学习Java提供的所有高级特性。本书将详细介绍下列内容：

- 面向对象程序设计
- 异常处理
- 反射与代理
- 泛型程序设计
- 接口与内部类
- 集合框架
- 事件监听器模型
- 并行操作
- 使用Swing UI工具箱进行图形用户界面设计

随着Java类库的爆炸式增长，一本书无法涵盖程序员需要了解的所有Java特性。因此，我们决定将本书分为两卷。卷I（本书）集中介绍Java的基本概念以及图形用户界面程序设计的基础知识。卷Ⅱ——高级特性，涉及企业特性以及高级的用户界面程序设计。其中包含下列内容：

- 文件与流
- 数据库
- 分布式对象
- 高级GUI组件

- 本地方法
- XML处理
- 网络编程
- 高级图形
- 国际化
- JavaBean
- 注释

在这一版中，我们对两卷中的内容进行了调整，特别是，鉴于多线程的日趋重要，我们将它编入卷I中，并以摩尔定律作为结尾。

在编写本书的过程中，难免出现错误和不准确之处。我们很想知道这些错误，当然，也希望同一个问题只被告知一次。我们在网页http://horstmann.coni/coreJava.html中以列表的形式给出了常见的问题、bug修正和出错位置。在勘误页（建议先阅读一遍）最后附有用来报告bug并提出修改意见的表单。如果我们不能回答每一个问题或没有及时回复，请不要失望。我们会认真地阅读所有的来信，感谢您的建议使本书后续的版本更清晰、更有指导价值。

关于本书

第1章概述Java与其他程序设计语言不同的性能。解释这种语言的设计初衷，以及在哪些方面达到了预期的效果。然后，简要叙述Java诞生和发展的历史。

第2章详细地论述如何下载和安装JDK以及本书的程序示例。然后，通过编译和运行三个典型的Java程序（一个控制台应用、一个图形应用、一个applet），指导读者使用简易的JDK、可启用Java的文本编辑器以及一个Java IDE。

第3章开始讨论Java 语言。这一章涉及的基础知识有变量、循环以及简单的函数。对于C或C++程序员来说，学习这一章的内容将会感觉一帆风顺，因为这些语言特性的语法本质上与C语言相同。对于没有C语言程序设计背景，但使用过其他程序设计语言（Visual Basic）的程序员，仔细地阅读这一章是非常必要的。

第4章介绍面向对象程序设计（Object-Oriented Programming, OOP）是当今程序设计的主流，而Java 是完全面向对象的。本章将介绍面向对象两个基本成分中最重要的——封装，以及Java 语言实现封装的机制，即类与方法。除了Java 语言规则之外，还对如何正确地进行OOP设计给出了忠告。最后，介绍奇妙的Javadoc 工具，它将代码注释转换为超链接的网页。熟悉C++的程序员可以快速地浏览这一章，而没有面向对象程序设计背景的程序员，应在进一步学习Java之前花一些时间了解OOP的有关概念。

第5章介绍类与封装仅仅是OOP中的一部分，本章将介绍另一部分——继承。继承使程序员可以使用现有的类，并根据需要进行修改。这是Java程序设计中的基础。Java中的继承机制与C++的继承机制十分相似。C++程序员只需关注两种语言的不同之处即可。

第6章展示如何使用Java的接口。接口可以让你的理解超越第5章的简单继承模型。掌握接口的使用将可以获得Java完全的面向对象程序设计的能力。本章还将介绍Java的一个有用的技术特性——内部类。内部类可以使代码更清晰、更简洁。

第7章开始细致地讨论应用程序设计。每一个Java程序员都应该了解一些图形用户界面程序设计的知识，本卷中包含了其中的基本内容部分。本章将展示如何制作窗口、如何在窗口中绘图、如何用几何图形作画、如何用多种字体格式化文本以及如何显示图像。

第8章详细讨论AWT（Abstract Window Toolkit ）的事件模型。我们将介绍如何编写代码来响

应鼠标点击或敲击键盘等事件。同时，还将介绍如何处理基本的GUI元素，比如：按钮和面板。

第9章详细讨论Swing GUI 工具箱。Swing工具箱允许建立一个跨平台的图形用户界面。本章将介绍如何建立各种各样的按钮、文本组件、边界、滑块、列表框、菜单以及对话框等等。一些更高级的组件将在卷II中讨论。

第10章阐述如何部署自己编写的应用程序或applet。在这里将描述如何将应用程序打包到JAR 文件中，以及如何使用Java的Web Start 机制在Internet上发布应用程序。最后，将解释Java程部署之后如何存储、检索配置信息。

第11章讨论异常处理，即Java的健壮机制，它用于处理调试好的程序可能出现的意外的情况。异常提供了一种将正常的处理代码与错误处理代码分开的有效手段。当然，即使程序包含处理所有异常情况的功能，依然有可能无法按照预计的方式工作。这一章的后半部分，将给出大量的实用调试技巧。最后，讲述如何使用各种工具完成一个示例程序。

第12章概要介绍泛型程序设计，这是Java SE5.0的一项重要改进。泛型程序设计使得程序拥有更好的可阅读性和安全性。在这里，将展示如何使用强类型机制，而舍弃不安全的强制类型转换，以及如何处理与旧版本Java兼容而带来的复杂问题。

第13章介绍Java平台的集合框架。当需要将大量对象收集到一起，并在过后要对它们进行检索时，可能会想要使用集合，这是目前最为合适的，它取代了将这些元素放置在数组中。本章将介绍如何使用预先建立好的标准集合。

第14章是本书的最后一章。在这章中，将介绍多线程，这是一种可以让程序任务并行执行的特性（线程是程序中的控制流），并阐述如何建立线程、如何处理线程的同步问题。从Java SE 5.0开始，多线程有了很大的改进，本章将介绍所有这些新的机制。

附录列出了Java语言的保留字。

约定

本书使用以下图标表示特殊内容。

✔ **注释**：这个图标表示为正文提供的“注释”信息。

! **提示**：这个图标表示为正文提供的“提示”信息。

✗ **警告**：这个图标表示为正文提供的“警告”信息。

C++ **C++注释**：在本书中有许多用来解释Java与C++的不同的C++注释。对于没有C++程序设计背景，或者不擅长C++程序设计的程序员，可以跳过这些注释。

API 应用程序编程接口

Java带有一个很大的程序设计库，即应用程序编程接口（API）。第一次调用API时，将会在这一节的结尾给出一个概要描述，并标有API图标。这些描述十分通俗易懂，希望能够比联机API文档提供更多的信息。我们在每一个API注释特性上都标记了版本号，用来提示那些不希

望使用Java“风险”版本的读者。

程序源代码按照下列格式给出：

Listing 2-4 `WelcomeApplet.Java`

示例代码

本书的网站http://www.phptr.conl/coreJava 以压缩的形式提供了书中的所有示例代码。可以用相应的解压缩程序或者用Java 开发工具箱中的jar实用程序展开这个文件。有关安装Java 开发工具箱和示例程序的详细信息请参看第2章。

致　谢

写一本书需要投入大量的精力，改写一本书也并不是想像的那样轻松，尤其是Java一直在持续不断地更新。编著一本书让很多人耗费了很多心血，在此衷心地感谢《Java核心技术》编写小组的每一位成员。

Prentice Hall和Sun Microsystems出版公司的许多人提供了非常有价值的帮助，却甘愿做幕后英雄。在此，我希望每一位都能够知道我对他们努力的感恩。与以往一样，我要真诚地感谢我的编辑，Prentice Hall公司的Greg Doench，从本书的写作到出版一直给予了指导，同时感谢那些不知其姓名的为本书做出贡献的幕后人士。感谢Vanessa Moore提供了优秀的产品支持。我还要感谢早期版本中我的合作者，Gary Cornell，他已经转向其他的事业。

感谢早期版本的许多读者，他们指出了许多令人尴尬的错误并给出了许多具有建设性的修改意见。我还要特别感谢本书优秀的审阅小组，他们仔细地审阅我的手稿，使本书减少了许多错误。

本书及早期版本的审阅专家包括：Chuck Allison（《C/C++Users Journal》的特约编辑），Alec Beaton (PointBase, Inc.), Cliff Berg (iSavvix Corporation), Joshua Bloch (Sun Microsystems), David Brown, Corky Cartwright, Frank Cohen (PushToTest), Chris Crane (devXsolution), Dr. Nicholas J. De Lillo (曼哈顿学院), Rakesh Dhoopar (Oracle), David Geary (Sabreware), Brian Goetz (Quiotix公司首席顾问), Angela Gordon (Sun Microsystems), Dan Gordon (Sun Microsystems), Rob Gordon, John Gray (哈特福大学), Cameron Gregory (olabs.com), Marty Hall (约翰斯·霍普金斯大学应用物理实验室), Vincent Hardy (Sun Microsystems), Dan Harkey (圣约瑟州立大学), William Higgins(IBM), Vladimir Ivanovic (PointBase),Jerry Jackson (ChannelPoint Software), Tim Kimmet (Preview Systems), Chris Laffra, Charlie Lai (Sun Microsystems), Angelika Langer, Doug Langston, Hang Lau (麦吉尔大学), Mark Lawrence, Doug Lea (SUNY Oswego), Gregory Longshore, Bob Lynch (Lynch Associates), Philip Milne (顾问), Mark Morrissey (俄勒冈研究院), Mahesh Neelakanta (佛罗里达大西洋大学), Hao Pham, Paul Philion, Blake Ragsdell, Stuart Reges (亚利桑那大学), Rich Rosen (Interactive Data Corporation), Peter Sanders (ESSI 大学, Nice, France), Dr. Paul Sanghera (圣约瑟州立大学与布鲁克斯学院), Paul Sevinc (Teamup AG), Devang Shah (Sun Microsystems), Bradley A. Smith, Steven Stelting (Sun Microsystems), Christopher Taylor, Luke Taylor (Valtech), George Thiruvathukal, Kim Topley (《Core JFC》的作者), Janet Traub, Paul Tyma (顾问), Peter van der Linden (Sun Microsystems), and Burt Walsh.

Cay Horstmann

2007于旧金山

目　录

译者序

前言

致谢

第1章　Java程序设计概述 ……1

1.1　Java程序设计平台 ……1

1.2　Java"白皮书"的关键术语 ……2

1.2.1　简单性 ……2

1.2.2　面向对象 ……3

1.2.3　网络技能 ……3

1.2.4　健壮性 ……3

1.2.5　安全性 ……4

1.2.6　体系结构中立 ……4

1.2.7　可移植性 ……4

1.2.8　解释型 ……5

1.2.9　高性能 ……5

1.2.10　多线程 ……5

1.2.11　动态性 ……6

1.3　Java Applet与Internet ……6

1.4　Java发展简史 ……7

1.5　关于Java的常见误解 ……9

第2章　Java程序设计环境 ……12

2.1　安装Java开发工具箱 ……12

2.1.1　下载JDK ……12

2.1.2　设置执行路径 ……13

2.1.3　安装源代码库和文档 ……15

2.1.4　安装本书中的示例 ……16

2.1.5　导航Java目录 ……16

2.2　选择开发环境 ……17

2.3　使用命令行工具 ……17

2.4　使用集成开发环境 ……20

2.5　运行图形化应用程序 ……22

2.6　建立并运行applet ……24

第3章　Java基本的程序设计结构 ……28

3.1　一个简单的Java应用程序 ……28

3.2　注释 ……31

3.3　数据类型 ……31

3.3.1　整型 ……32

3.3.2　浮点类型 ……32

3.3.3　char类型 ……33

3.3.4　boolean类型 ……35

3.4　变量 ……35

3.4.1　变量初始化 ……36

3.4.2　常量 ……36

3.5　运算符 ……37

3.5.1　自增运算符与自减运算符 ……38

3.5.2　关系运算符与boolean运算符 ……38

3.5.3　位运算符 ……39

3.5.4　数学函数与常量 ……40

3.5.5　数值类型之间的转换 ……41

3.5.6　强制类型转换 ……41

3.5.7　括号与运算符级别 ……42

3.5.8　枚举类型 ……43

3.6　字符串 ……43

3.6.1　子串 ……43

3.6.2　拼接 ……44

3.6.3　不可变字符串 ……44

3.6.4　检测字符串是否相等 ……45

3.6.5　代码点与代码单元 ……46

3.6.6　字符串API ……47

3.6.7　阅读联机API文档 ……48

3.6.8　构建字符串 ……50

3.7　输入输出 ……51

3.7.1　读取输入 ……52

3.7.2 格式化输出 ……54
3.7.3 文件输入与输出 ……57
3.8 控制流程 ……58
3.8.1 块作用域 ……59
3.8.2 条件语句 ……59
3.8.3 循环 ……62
3.8.4 确定循环 ……65
3.8.5 多重选择：switch语句 ……68
3.8.6 中断控制流程语句 ……70
3.9 大数值 ……72
3.10 数组 ……74
3.10.1 For each循环 ……75
3.10.2 数组初始化以及匿名数组 ……76
3.10.3 数组拷贝 ……76
3.10.4 命令行参数 ……78
3.10.5 数组排序 ……79
3.10.6 多维数组 ……82
3.10.7 不规则数组 ……84
第4章 对象与类 ……87
4.1 面向对象程序设计概述 ……87
4.1.1 类 ……88
4.1.2 对象 ……89
4.1.3 识别类 ……89
4.1.4 类之间的关系 ……90
4.2 使用现有类 ……91
4.2.1 对象与对象变量 ……91
4.2.2 Java类库中的GregorianCalendar类 ……94
4.2.3 更改器方法与访问器方法 ……95
4.3 用户自定义类 ……101
4.3.1 一个Employee类 ……101
4.3.2 多个源文件的使用 ……104
4.3.3 解析Employee类 ……104
4.3.4 从构造器开始 ……105
4.3.5 隐式参数与显式参数 ……106
4.3.6 封装的优点 ……107
4.3.7 基于类的访问权限 ……109
4.3.8 私有方法 ……109
4.3.9 Final实例域 ……110
4.4 静态域与静态方法 ……110
4.4.1 静态域 ……110
4.4.2 静态常量 ……111
4.4.3 静态方法 ……111
4.4.4 Factory方法 ……112
4.4.5 Main方法 ……113
4.5 方法参数 ……115
4.6 对象构造 ……120
4.6.1 重载 ……120
4.6.2 默认域初始化 ……121
4.6.3 默认构造器 ……121
4.6.4 显式域初始化 ……122
4.6.5 参数名 ……123
4.6.6 调用另一个构造器 ……123
4.6.7 初始化块 ……124
4.6.8 对象析构与finalize方法 ……127
4.7 包 ……128
4.7.1 类的导入 ……128
4.7.2 静态导入 ……130
4.7.3 将类放入包中 ……130
4.7.4 包作用域 ……133
4.8 类路径 ……134
4.9 文档注释 ……136
4.9.1 注释的插入 ……136
4.9.2 类注释 ……137
4.9.3 方法注释 ……137
4.9.4 域注释 ……138
4.9.5 通用注释 ……138
4.9.6 包与概述注释 ……139
4.9.7 注释的抽取 ……140
4.10 类设计技巧 ……140
第5章 继承 ……143
5.1 类、超类和子类 ……143
5.1.1 继承层次 ……149
5.1.2 多态 ……149
5.1.3 动态绑定 ……151

5.1.4 阻止继承：final类和方法 ……152
5.1.5 强制类型转换 ……154
5.1.6 抽象类 ……155
5.1.7 受保护访问 ……160
5.2 Object：所有类的超类 ……160
5.2.1 Equals方法 ……161
5.2.2 相等测试与继承 ……162
5.2.3 HashCode方法 ……164
5.2.4 ToString方法 ……166
5.3 泛型数组列表 ……171
5.3.1 访问数组列表元素 ……173
5.3.2 类型化与原始数组列表的兼容性 ……176
5.4 对象包装器与自动打包 ……177
5.5 参数数量可变的方法 ……179
5.6 枚举类 ……181
5.7 反射 ……182
5.7.1 Class类 ……183
5.7.2 捕获异常 ……184
5.7.3 利用反射分析类的能力 ……186
5.7.4 在运行时使用反射分析对象 ……191
5.7.5 使用反射编写泛型数组代码 ……195
5.7.6 方法指针 ……198
5.8 继承设计的技巧 ……201
第6章 接口与内部类 ……204
6.1 接口 ……204
6.1.1 接口的特性 ……209
6.1.2 接口与抽象类 ……210
6.2 对象克隆 ……211
6.3 接口与回调 ……216
6.4 内部类 ……219
6.4.1 使用内部类访问对象状态 ……220
6.4.2 内部类的特殊语法规则 ……223
6.4.3 内部类是否有用、必要和安全 ……224
6.4.4 局部内部类 ……226
6.4.5 由外部方法访问final变量 ……226
6.4.6 匿名内部类 ……229
6.4.7 静态内部类 ……231
6.5 代理 ……234
第7章 图形程序设计 ……239
7.1 Swing概述 ……239
7.2 创建框架 ……242
7.3 框架定位 ……244
7.4 框架属性 ……246
7.5 决定框架大小 ……246
7.6 在组件中显示信息 ……249
7.7 2D图形 ……253
7.8 颜色 ……260
7.9 为文本设定特殊字体 ……263
7.10 图像 ……270
第8章 事件处理 ……274
8.1 事件处理基础 ……274
8.1.1 实例：处理按钮点击事件 ……276
8.1.2 建议使用内部类 ……280
8.1.3 创建包含一个方法调用的监听器 ……282
8.1.4 实例：改变观感 ……283
8.1.5 适配器类 ……286
8.2 动作 ……290
8.3 鼠标事件 ……296
8.4 AWT事件继承层次 ……302
第9章 Swing用户界面组件 ……306
9.1 Swing和模型-视图-控制器设计模式 ……306
9.1.1 设计模式 ……306
9.1.2 模型-视图-控制器模式 ……307
9.1.3 Swing按钮的模型-视图-控制器分析 ……310
9.2 布局管理器概述 ……311
9.2.1 边框布局 ……313
9.2.2 网格布局 ……314
9.3 文本输入 ……318
9.3.1 文本域 ……319
9.3.2 标签和标签组件 ……320
9.3.3 密码域 ……321
9.3.4 文本区 ……322
9.3.5 滚动窗格 ……322
9.4 选择组件 ……325

9.4.1 复选框 ……325
9.4.2 单选按钮 ……327
9.4.3 边框 ……331
9.4.4 组合框 ……335
9.4.5 滑块 ……338
9.5 菜单 ……344
9.5.1 菜单创建 ……344
9.5.2 菜单项中的图标 ……346
9.5.3 复选框和单选按钮菜单项 ……347
9.5.4 弹出菜单 ……348
9.5.5 快捷键和加速器 ……349
9.5.6 启用和禁用菜单项 ……351
9.5.7 工具栏 ……355
9.5.8 工具提示 ……356
9.6 复杂的布局管理 ……359
9.6.1 网格组布局 ……360
9.6.2 组布局 ……369
9.6.3 不使用布局管理器 ……377
9.6.4 定制布局管理器 ……378
9.6.5 遍历顺序 ……382
9.7 对话框 ……383
9.7.1 选项对话框 ……383
9.7.2 创建对话框 ……392
9.7.3 数据交换 ……396
9.7.4 文件对话框 ……402
9.7.5 颜色选择器 ……412
第10章 部署应用程序和applet ……418
10.1 JAR文件 ……418
10.1.1 清单文件 ……419
10.1.2 可运行JAR文件 ……420
10.1.3 资源 ……421
10.1.4 密封 ……423
10.2 Java Web Start ……424
10.2.1 沙箱 ……427
10.2.2 签名代码 ……428
10.2.3 JNLP API ……430
10.3 Applet ……437
10.3.1 一个简单的applet ……438
10.3.2 将应用程序转换为applet ……440
10.3.3 Applet的HTML标记和属性 ……441
10.3.4 Object标记 ……444
10.3.5 使用参数向applet传递信息 ……444
10.3.6 访问图像和音频文件 ……449
10.3.7 Applet上下文 ……450
10.4 应用程序存储的配置 ……457
10.4.1 属性映射 ……457
10.4.2 Preferences API ……462
第11章 异常、日志、断言和调试 ……468
11.1 处理异常 ……468
11.1.1 异常分类 ……470
11.1.2 声明已检查异常 ……471
11.1.3 如何抛出异常 ……473
11.1.4 创建异常类 ……474
11.2 捕获异常 ……475
11.2.1 捕获多个异常 ……477
11.2.2 再次抛出异常与异常链 ……477
11.2.3 Finally子句 ……478
11.2.4 分析堆栈跟踪元素 ……481
11.3 使用异常机制的建议 ……483
11.4 断言 ……486
11.4.1 启用和禁用断言 ……487
11.4.2 使用断言的建议 ……487
11.4.3 为文档使用断言 ……488
11.5 记录日志 ……489
11.5.1 基本日志 ……490
11.5.2 高级日志 ……490
11.5.3 修改日志管理器配置 ……492
11.5.4 本地化 ……493
11.5.5 处理器 ……494
11.5.6 过滤器 ……496
11.5.7 格式化器 ……497
11.5.8 日志记录说明 ……497
11.6 调试技术 ……505
11.6.1 使用控制台窗口 ……510

11.6.2 跟踪AWT事件 …………………………511
11.6.3 AWT的Robot类…………………………515
11.7 使用调试器 ……………………………………519
第12章 泛型程序设计………………………………523
12.1 为什么要使用泛型程序设计 ……………523
12.2 简单泛型类的定义 …………………………525
12.3 泛型方法 ………………………………………526
12.4 类型变量的限定 ……………………………527
12.5 泛型代码和虚拟机 …………………………529
12.5.1 翻译泛型表达式 ……………………531
12.5.2 翻译泛型方法 ………………………531
12.5.3 调用遗留代码 ………………………533
12.6 约束与局限性 …………………………………534
12.6.1 不能用基本类型实例化类型参数 …534
12.6.2 运行时类型查询只适用于原始类型…534
12.6.3 不能抛出也不能捕获泛型类实例 …535
12.6.4 参数化类型的数组不合法 …………535
12.6.5 不能实例化类型变量 ………………536
12.6.6 泛型类的静态上下文中类型变量无效 ……………………………537
12.6.7 注意擦除后的冲突 …………………537
12.7 泛型类型的继承规则 ………………………538
12.8 通配符类型 ……………………………………540
12.8.1 通配符的超类型限定 ………………541
12.8.2 无限定通配符 …………………………544
12.8.3 通配符捕获 ……………………………544
12.9 反射和泛型 ……………………………………547
12.9.1 使用Class<T>参数进行类型匹配 …548
12.9.2 虚拟机中的泛型类型信息 …………549
第13章 集合 ……………………………………………554
13.1 集合接口 ………………………………………554
13.1.1 将集合的接口与实现分离 …………554
13.1.2 Java类库中的集合接口和迭代器接口…557
13.2 具体的集合 ……………………………………561
13.2.1 链表 ……………………………………562
13.2.2 数组列表 ………………………………570
13.2.3 散列集 …………………………………570
13.2.4 树集 ……………………………………573
13.2.5 对象的比较 ……………………………574
13.2.6 队列与双端队列 ……………………579
13.2.7 优先级队列 ……………………………580
13.2.8 映射表 …………………………………581
13.2.9 专用集与映射表类 …………………585
13.3 集合框架 ………………………………………589
13.3.1 视图与包装器 …………………………592
13.3.2 批操作 …………………………………598
13.3.3 集合与数组之间的转换 ……………599
13.4 算法 ……………………………………………599
13.4.1 排序与混排 ……………………………601
13.4.2 二分查找 ………………………………603
13.4.3 简单算法 ………………………………604
13.4.4 编写自己的算法 ……………………605
13.5 遗留的集合 ……………………………………606
13.5.1 Hashtable 类 …………………………606
13.5.2 枚举 ……………………………………607
13.5.3 属性映射表 ……………………………608
13.5.4 栈 ………………………………………608
13.5.5 位集 ……………………………………609
第14章 多线程 …………………………………………613
14.1 线程的概念 ……………………………………613
14.2 中断线程 ………………………………………623
14.3 线程状态 ………………………………………625
14.3.1 新生线程 ………………………………626
14.3.2 可运行线程 ……………………………626
14.3.3 被阻塞线程和等待线程 ……………626
14.3.4 被终止的线程 …………………………627
14.4 线程属性 ………………………………………628
14.4.1 线程优先级 ……………………………628
14.4.2 守护线程 ………………………………629
14.4.3 未捕获异常处理器 …………………629
14.5 同步 ……………………………………………631
14.5.1 竞争条件的一个例子 ………………631
14.5.2 详解竞争条件 …………………………635
14.5.3 锁对象 …………………………………636

14.5.4 条件对象 ……639
14.5.5 synchronized关键字 ……643
14.5.6 同步阻塞 ……646
14.5.7 监视器概念 ……647
14.5.8 Volatile域 ……648
14.5.9 死锁 ……649
14.5.10 锁测试与超时 ……652
14.5.11 读/写锁 ……653
14.5.12 为什么弃用stop和suspend方法 ……654
14.6 阻塞队列 ……655
14.7 线程安全的集合 ……661
14.7.1 高效的映像、集合和队列 ……662
14.7.2 写数组的拷贝 ……663
14.7.3 旧的线程安全的集合 ……663
14.8 Callable与Future ……664
14.9 执行器 ……668
14.9.1 线程池 ……669
14.9.2 预定执行 ……673
14.9.3 控制任务组 ……673
14.10 同步器 ……675
14.10.1 信号量 ……675
14.10.2 倒计时门栓 ……675
14.10.3 障栅 ……676
14.10.4 交换器 ……676
14.10.5 同步队列 ……677
14.10.6 例子：暂停动画与恢复动画 ……677
14.11 线程与Swing ……682
14.11.1 运行耗时的任务 ……683
14.11.2 使用Swing工作器 ……687
14.11.3 单一线程规则 ……693

第1章　Java程序设计概述

▲ Java程序设计平台
▲ Java“白皮书”的关键术语
▲ Java与Internet
▲ Java发展简史
▲ 关于Java的常见误解

1996年Java第一次发布就引起了人们的极大兴趣。关注Java的人士不仅限于计算机出版界，还有诸如《纽约时报》、《华盛顿邮报》、《商业周刊》这样的主流媒体。Java是第一种也是惟一的一种在National Public Radio上占用了10分钟时间进行介绍的程序设计语言，并且还得到了$100 000 000的风险投资基金。这些基金全部用来支持用这种特别的计算机语言开发的产品。重温那些令人兴奋的日子是很有意思的。本章将简要地介绍一下Java语言的发展历史。

1.1　Java程序设计平台

本书的第1版是这样描写Java的：“作为一种计算机语言，Java的广告词确实有点夸大其辞。然而，Java的确是一种优秀的程序设计语言。作为一个名副其实的程序设计人员，使用Java无疑是一个好的选择。有人认为：Java将有望成为一种最优秀的程序设计语言，但还需要一个相当长的发展时期。一旦一种语言应用于某个领域，与现存代码的相容性问题就摆在了人们的面前。”

我们的编辑手中有许多这样的广告词。这是Sun公司高层的某位不愿透露姓名的人士提供的。然而，现在看起来，当初的这些预测还是有一定准确性的。Java有许多非常优秀的语言特性，本章稍后将会详细地讨论这些特性。由于相容性这个严峻的问题确实存在于现实中，所以，或多或少地还是有一些“累赘”被加到语言中，这就导致Java并不如想像中的那么完美无瑕。

但是，正像我们在第1版中已经指出的那样，Java并不只是一种语言。在此之前出现的那么多种语言也没有能够引起那么大的轰动。Java是一个完整的平台，有一个庞大的库，其中包含了很多可重用的代码和一个提供诸如安全性、跨操作系统的可移植性以及自动垃圾收集等服务的执行环境。

作为一名程序设计人员，常常希望能够有一种语言，它具有令人赏心悦目的语法和易于理解的语义（C++不是这样的）。与许多其他的优秀语言一样，Java恰恰满足了这些要求。有些语言提供了可移植性、垃圾收集器等等，但是，没有提供一个大型的库。如果想要有奇特的绘图功能、网络连接功能和数据库存取功能就必须自己动手编写代码。Java这种功能齐全的出色语言，具有高质量的执行环境以及庞大的库。正是因为它集多种优势于一身，所以对广大的程序设计人员有着不可抗拒的吸引力。

1.2 Java“白皮书”的关键术语

Java的设计者已经编写了颇有影响力的“白皮书”，用来解释设计的初衷以及完成的情况，并且发布了一个简短的摘要。这个摘要用下面11个关键术语进行组织：

简单性	可移植性
面向对象	解释型
网络技能（Network-Savvy）	高性能
健壮性	多线程
安全性	动态性
体系结构中立	

本节将论述下列主要内容：

- 给出白皮书中对每个关键术语的概述，这是Java设计者对相关术语的论述。
- 凭借Java当前版本的使用经验，给出对这些术语的理解。

注释： 白皮书可以在http://java.sun.com/docs/white/langenv/上找到。对于11个关键术语的论述请参看http://java.sun.com/docs/overviews/java/java-overview-1.html。

1.2.1 简单性

> 人们希望构建一个无需深奥的专业训练就可以进行编程的系统，并且要符合当今的标准惯例。因此，尽管人们发现C++不太适用，但在设计Java的时候还是尽可能地接近C++，以便系统更易于理解。Java剔除了C++中许多很少使用、难以理解、易混淆的特性。在目前看来，这些特性带来的麻烦远远多于其带来的好处。

的确，Java语法是C++语法的一个“纯净”版本。这里没有头文件、指针运算（甚至指针语法）、结构、联合、操作符重载、虚基类等等（请参阅本书各个章节给出的C++注释，那里比较详细地解释了Java与C++之间的区别）。然而，设计者并没有试图清除C++中所有不适当的特性。例如，`switch`语句的语法在Java中就没有改变。如果知道C++就会发现可以轻而易举地将其转换成Java。

如果已经习惯于使用可视化的编程环境（例如Visual Basic），你就不会觉得Java简单了。Java有许多奇怪的语法（尽管掌握其要领并不需要很长时间），更重要的是，使用Java需要自己编写大量的程序。Visual Basic的魅力在于它的可视化设计环境几乎自动地为应用程序提供了大量的基础结构。而使用Java实现同样的功能却需要手工地编制代码，通常代码量还相当大。然而，已经有一些支持“拖放”风格程序开发的第三方开发环境。

> 简单的另一个方面是小。Java的目标之一是支持开发能够在小型机器上独立运行的软件。基本的解释器以及类支持大约仅为40KB；再加上基础的标准类库和对线程的支持（基本上是一个自包含的微内核）大约需要增加175KB。

在当时，这是一个了不起的成就。当然，由于不断的扩展，类库已经相当庞大了。现在有一个独立的具有较小类库的Java微型版（Java Micro Edition）用于嵌入式设备。

1.2.2 面向对象

> 简单地讲，面向对象设计是一种程序设计技术。它将重点放在数据（即对象）和对象的接口上。用木匠打一个比方，一个“面向对象的”木匠始终关注的是所制作的椅子，第二位才是所使用的工具；一个“非面向对象的”木匠首先考虑的是所用的工具。在本质上，Java的面向对象能力与C++是一样的。

在过去的30年里，面向对象已经证明了自身的价值，一种现代的程序设计语言不使用面向对象技术简直让人难以置信。的确，Java的面向对象特性与C++旗鼓相当。Java与C++的主要不同点在于多继承，在Java中，取而代之的是简单的接口概念，以及Java的元类（metaclass）模型（有关这部分内容将在第5章中讨论）。



注释：如果没有使用面向对象程序设计语言的经验，你一定要仔细阅读第4章~第6章。这些章节解释了什么是面向对象程序设计以及在编程实现复杂的项目时为什么比传统的像C或Basic这样的面向过程的语言更加有效。

1.2.3 网络技能

> Java有一个扩展的例程库，用于处理像HTTP和FTP这类的TCP/IP协议。Java应用程序能够通过URL打开和访问网络上的对象，其便捷程度就好像访问本地文件一样。

人们已经看到Java的网络能力强大且易于使用。任何曾经试图使用其他语言进行网络编程的人都会惊呼Java竟然把类似打开socket连接这类繁重的任务都变得如此简单（在本书的卷II中介绍网络连接）。另外，远程方法调用机制使得分布式对象之间可以进行通信（也将在卷II中介绍）。

1.2.4 健壮性

> Java的设计目标之一在于使得Java编写的程序具有多方面的可靠性。Java投入了大量的精力进行早期的问题检测、后期动态的（运行时）检测，并消除了有出错倾向的状态 ...... Java和C++最大的不同在于Java采用的指针模型可以消除重写内存和损坏数据的可能性。

这个特性非常有用。Java编译器能够检测许多在其他语言中仅在运行时刻才能够检测出来的问题。至于第二点，对于曾经花费几个小时来检查由于指针bug而引起内存冲突的人来说，一定很喜欢Java的这一特性。

如果曾经只使用过Visual Basic这类没有显式指针的语言，你就会感觉这么说似乎有些小题大做了。然而，C程序员就没有这样幸运了。他们需要利用指针存取字符串、数组、对象、甚至文件。在Visual Basic中，根本不必使用指针访问这些实体，也不必关心有关内存分配的问题。另一方面，在没有指针的语言中，许多数据结构很难实现。Java具有双方的优势。它不需要使用指针构造诸如字符串、数组这样的结构。如果必要的话，它也能够具有指针的能力，如链表。Java绝对是安全的，其原因是永远不会存取一个“坏的”指针，造成内存分配的错误，也不必防范内存泄漏。

1.2.5 安全性

> Java适用于网络/分布式环境。为了达到这个目标，在安全方面投入了很大精力。使用Java可以构建防病毒、防篡改的系统。

本书的第1版曾经说过："永远不要把话说绝！"。事实证明这是正确的。在Java开发工具箱第1版启用后不久，普林斯顿大学的一些安全专家们就发现了在JDK1.0中的某些安全特性方面存在着一些非常隐蔽的bug。Sun Microsystems 大力支持对Java的安全性的研究，制定了供人们使用的规范，实现了虚拟机和安全库，并迅速地处理了所有已知的安全bug。在任何情况下，蒙骗Java的安全机制都是十分困难的。现在，发现bug 的技术越来越强，数目越来越少。

从一开始，Java就设计成能够防范各种袭击，其中包括：

- 运行时堆栈溢出。如，蠕虫等病毒常用的袭击手段。
- 在自己的处理空间之外破坏内存。
- 未经授权读写文件。

许多安全特性相继不断地加入到Java中。自从Java1.1问世以来，Java就有了数字签名类(digitally signed class)的概念（请参看卷II)。通过数字签名类，可以确定类的作者。如果信任这个类的作者，这个类就可以在机器上拥有更多的权限。



注释：来自微软的基于ActiveX技术的竞争代码传输机制，其安全性完全依赖于数字签名。这显然是不够的，因为微软自身产品的任何用户都可以证实，来自知名提供商的程序会崩溃并对系统产生危害。Java的安全机制比ActiveX要强得多，因为它是在应用程序运行时加以控制并制止恶意性破坏的。

1.2.6 体系结构中立

> 编译器生成一个体系结构中立的目标文件格式，这是一种编译过的代码，只要有Java运行时系统，就可以在许多处理器上运行。Java编译器通过生成与特定的计算机体系结构无关的字节码指令来实现这一特性。精心设计的字节码不仅可以很容易地在任何机器上解释执行，而且还可以迅速地翻译成本地机器的代码。

这并不是什么新的思路。30多年以前，Niklaus Wirth 实现的原始 Pascal以及UCSD Pascal系统都使用了这种技术。

当然，解释字节码肯定会比全速地运行机器指令慢很多。所以说，这是不是一个好的思路还很难讲！然而，虚拟机有一个选项，可以将使用最频繁的字节码序列翻译成机器码，这一过程被称为即时编译。这一策略已经证明十分有效，致使微软的.NET平台也依赖于虚拟机。

虚拟机还有一些其他的优点。虚拟机可以检测指令序列的行为，以增强其安全性。有些程序还可以快速地生成字节码，并动态地增强所运行程序的处理能力。

1.2.7 可移植性

> 与C和C++不同，Java规范中没有"依赖具体实现"的地方。基本数据类型的大小以及有关算法都做了明确的说明。

例如，Java中的`int`永远为32位的整数，而在C/C++中，`int`可能是16位整数、32位整数，也可能是编译器提供商指定的其他大小。惟一的限制只是`int`类型的大小不能低于`short int`，并且不能高于`long int`。在Java中，数据类型具有固定的大小，这消除了代码移植时令人头痛的主要问题。二进制数据以固定的格式进行的存储和传输，消除了字节顺序的困扰。字符串是用标准的Unicode格式存储的。

> *作为系统组成部分的类库，定义了可移植的接口。例如，有一个抽象的`Window`类给出了在UNIX、Windows和Macintosh环境下的不同实现。*

凡是尝试过的人都知道，要编写一个在Windows、Macintosh和10种不同风格的、在UNIX上看起来都不错的程序有多么困难。Java 1.0就尝试着做了这么一个壮举，发布了一个将常用的用户界面元素映射到不同平台上的简单工具箱。遗憾的是，花费了大量的心血，却构建了一个在各个平台上都难以让人接受的库（而且，在不同平台的图形实现中有不同的bug）。不过，这毕竟是个开端。对于许多应用问题来说，可移植性比华而不实的用户界面更加重要；而且这些应用程序从Java的早期版本中获益匪浅。现在，用户界面工具箱已经完全重写了，不再依赖于主机的用户接口。现在的Java版本比早期版本更加稳定，更加吸引人。

1.2.8 解释型

> *Java解释器可以在任何移植了解释器的机器上执行Java字节码。由于链接是一个增值且简便的过程，所以，开发过程也变得更加快捷，更加具有探索性。*

增值链接有其优势，但给开发过程带来的好处显然是言过其实了。事实上，早期的Java开发工具的速度相当慢。现在，使用即时编译器将字节码翻译成机器码。

1.2.9 高性能

> *尽管对解释后的字节码性能已经比较满意，但在有些场合下却需要更加高效的性能。字节码可以（在运行时刻）快速地翻译成运行这个应用程序的特定CPU的机器码。*

使用Java 的头几年，许多用户不同意这样的看法：性能就是“适用性更强”。然而，现在的即时编译器已经非常出色了，以至于成为了传统编译器的竞争对手。在某些情况下，甚至超越了传统编译器，其原因是它们含有更多的可用信息。例如，即时编译器可以监控经常执行哪些代码并优化这些代码以提高速度。更为复杂的优化是消除函数调用（即“内嵌”）。即时编译器知道哪些类已经加载。如果基于当前加载的类集，且特定的函数不被覆盖的话就可以内嵌。必要时，还可以撤销优化。

1.2.10 多线程

> *多线程可以带来更好的交互响应和实时行为。*

如果曾经使用过其他语言编写多线程的应用程序，就会对Java多线程处理的便捷性惊叹不已。只要操作系统支持，Java中的线程就可以利用多个处理器。在底层，主流平台的线程实现机制各不相同，Java并没有花费太大的力气对此实现平台无关性。在不同的机器上，只是调用多线程的代码完全相同；Java把多线程的实现交给了底层的操作系统或线程库来完成。尽管如

此，多线程编译的简单性是Java成为颇具魅力的服务器端开发语言的主要原因之一。

1.2.11 动态性

> 从各种角度看，Java与C或C++相比更加具有动态性。它能够适应不断发展的环境。库中可以自由地添加新方法和实例变量，而对客户端却没有任何影响。在Java中找出运行时类型信息十分简单。

当需要将某些代码添加到正在运行的程序中时，动态性将是一个非常重要的特性。一个很好的例子是：从Internet上下载代码，然后在浏览器上运行。在Java 1.0中，不能直接获得运行时的类型信息，而Java的当前版本允许程序员知道对象的结构和行为。这对于必须在运行时分析对象的系统来说非常有用。这些系统有：Java GUI 构建器、智能调试器、可插入组件以及对象数据库。

注释：Java成功地推出后不久，微软就发布了一个叫做J++的产品，它与Java有相同的编程语言以及虚拟机。现在，微软不再支持J++，取而代之的是另一种被称为C# 的语言。C# 与Java有很多相似之处，然而使用的却是完全不同的虚拟机。甚至还有一种J# 语言可将J++ 的应用迁移到使用C# 的虚拟机上。本书不准备介绍J++、C# 或J# 语言。

1.3 Java Applet与Internet

这里的想法很简单：用户从Internet下载Java字节码，并在自己的机器上运行。在网页中运行Java程序称为applet。为了使用applet，需要启用Java的Web浏览器执行字节码。由于Sun公司负责发放Java源代码的许可证，并坚持不允许对语言和基本类库的结构做出任何修改，因此，Java的applet应该可以运行在任何启用Java浏览器上，并且无论何时访问包含applet的网页，都会得到程序的最终版本。最重要的是，要感谢虚拟机的安全机制，它让我们不必再担心来自恶意代码的袭击。

用户下载一个applet 就如同在网页中嵌入一幅图片。applet成为了页面的一部分。文本环绕着applet所占据的空间周围。关键一点是图片是活动的。它可以对用户命令做出响应，改变外观，在运行它的计算机与提供它的计算机之间传递数据。

图1-1展示了一个很好的动态网页的示例。Jmol applet显示了分子结构，这将需要相当复杂的计算。在这个网页中，可以利用鼠标进行旋转，调整焦距等操作，以便更加细致地理解分子结构。用静态网页将无法实现这种直接的操作，而applet却可以达到此目的。（可以在http://jmol.sourceforge.net上找到这个applet）

当applet首次出现时，人们欣喜若狂。许多人相信applet的魅力将会导致Java迅速地流行起来。然而，初期的兴奋很快就淡化了。不同版本的Netscape与Internet Explorer运行不同版本的Java，其中有些早已过时。这种糟糕的情况导致更加难于利用Java的最新版本开发applet。今天，当需要在浏览器中显示动态效果时，大多数网页都直接使用JavaScript或Flash。另外，Java已经成为用来开发服务器端应用程序的最流行的语言，使用这些服务器端应用程序可以产生网页、运行后端逻辑。

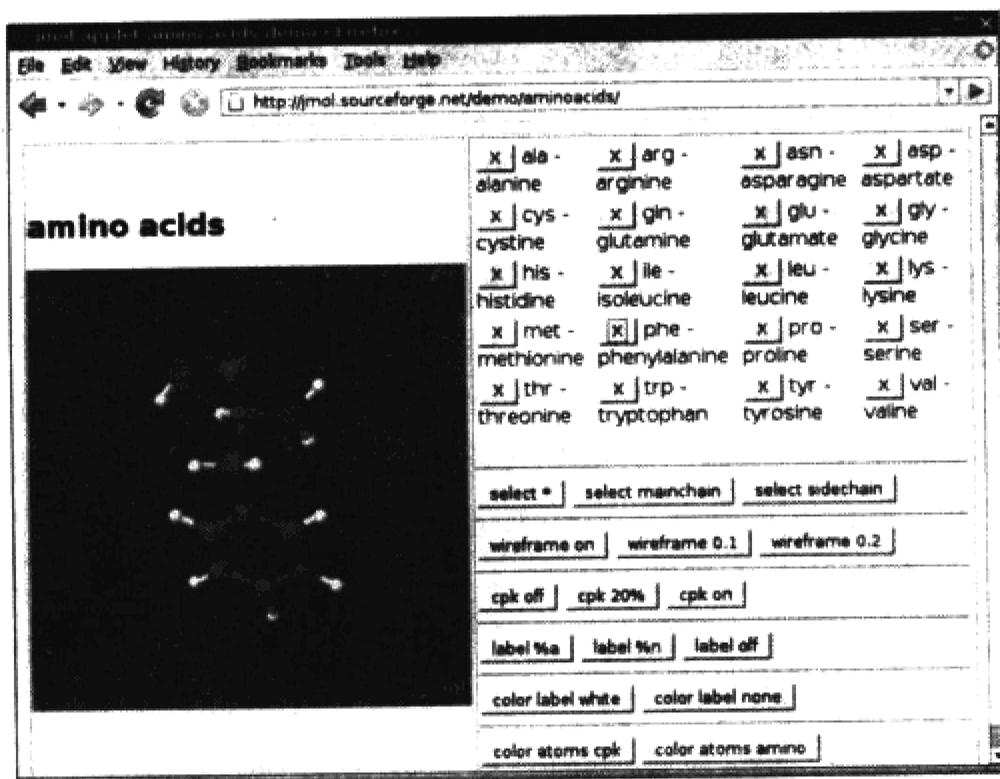


图1-1 Jmol applet

1.4 Java发展简史

本节将介绍Java的发展简史。这些参考资料来源于多方面的出版物（最重要的是SunWorld的在线杂志1995年7月上对Java创建者的专访）。

Java的历史要追溯到1991年，由Patrick Naughton及其伙伴James Gosling（一个全能的计算机奇才）带领的Sun公司的工程师小组想要设计一种小型的计算机语言，主要用于像有线电视转换盒这类的消费设备。由于这些消费设备的处理能力和内存都很有限，所以语言必须非常小且能够生成非常紧凑的代码。另外，由于不同的厂商会选择不同的中央处理器（CPU），因此这种语言的关键是不能与任何特定的体系结构捆绑在一起。这个项目被命名为“Green”。

代码短小、紧凑且与平台无关，这些要求促使开发团队联想起很早以前的一种模型，某些Pascal的实现曾经在早期的PC上尝试过这种模型。以Pascal的发明者Niklaus Wirth为先驱，率先设计出一种为假想的机器生成中间代码的可移植语言（假想的机器称为虚拟机——Java虚拟机即JVM的命名由此而来）。这种中间代码可以应用于所有已经正确安装解释器的机器上。Green项目工程师也使用了虚拟机，从而解决了课题中的主要问题。

不过，Sun公司的人都有UNIX的应用背景。因此，所开发的语言以C++为基础，而不是Pascal。特别是这种语言是面向对象的，而不是面向过程的。就像Gosling在专访中谈到的：“毕竟，语言只是实现目标的工具，而不是目标本身”。Gosling把这种语言称为“Oak”（这么起名的原因大概是因为他非常喜欢自己办公室外的橡树）。Sun公司的人后来发现Oak是一种已有的计算机语言的名字，于是，将其改名为Java。事实证明这是一个极好的选择。

1992年，Green项目发布了它的第一个产品，称之为“*7”。这个产品具有非常智能的远程控制（其能力相当于长6英寸、宽4英寸、高4英寸的SPARC工作站）。遗憾的是，Sun公司对生产这个产品并不感到兴趣，Green项目组的人员必须找出其他的方法来将他们的技术推向市场。然而，没有一个标准消费品公司对此感兴趣。于是，Green项目组竞标了一个提供视频点播等

新型服务的有线电视盒的项目，但没有成功（有趣的是，得到这个项目的公司的领导恰恰是开创Netscape公司的Jim Clark。Netscape公司后来对Java的成功给予了很大的帮助）。

Green项目（这时换了一个新名字——“First Person公司”）花费了1993年一整年以及1994年的上半年，一直在苦苦寻求其技术的买家。然而，一个也没有找到（Patrick Naughton，项目组的创立人之一，也是此项目经营工作收尾的人，声称为了销售这项技术，他们累计飞行了300 000英里）。1994年First Person公司解散了。

当这一切在Sun 公司中继续进行的时候，Internet的万维网也日渐发展壮大。Web 的关键是把超文本页面转换到屏幕上的浏览器。1994年大多数人都在使用Mosaic，这是一个1993年出自伊利诺斯大学超级计算中心的非商业化的Web浏览器（Mosaic的一部分是由Marc Andreessen编写的。当时，他作为一名参加半工半读项目的本科生，编写了这个软件，每小时的薪水只有6.85美元。他后来成为了Netscape公司的创始人之一和技术总监，可谓名利双收）。

在接受SunWorld 采访的时候，Gosling 说在1994年中期，Java语言的开发者意识到：“我们能够建立一个最酷的浏览器。我们已经拥有在客户机/服务器主流模型中所需要的体系结构中立、实时、可靠、安全——这些在工作站环境并不太重要，所以，我们决定开发浏览器。”

实际的浏览器是由Patrick Naughton和Jonathan Payne开发的，并演变为HotJava浏览器。为了炫耀Java语言超强的能力，HotJava浏览器采用Java编写。然而，设计者也非常清楚现在所谓的applet的威力，因此他们让HotJava浏览器具有执行网页中内嵌代码的能力。这一“技术印证”在1995年5月23日的SunWorld上得到展示，同时引发了人们延续至今的对Java的狂热追逐。

1996年初，Sun发布了Java的第1个版本。人们很快地意识到Java1.0不能用来进行真正的应用开发。的确，可以使用Java1.0来实现在画布上随机跳动的nervous文本，但它却没有提供打印功能。坦率地说，Java1.0的确没有为其黄金时期的到来做好准备。后来的Java1.1 弥补了其中的大多部分明显的缺陷，大大改进了反射能力，并为GUI编程增加了新的事件处理模型。尽管它仍然具有很大的局限性。

1998年JavaOne会议的头号新闻是即将发布Java1.2版。这个版本取代了早期玩具式的GUI，并且它的图形工具箱更加精细而具有可伸缩性，更加接近“一次编写，随处运行”的承诺。在1998年12月Java1.2发布三天之后，Sun公司市场部将其名称改为更加吸引人的“Java2标准版软件开发工具箱1.2版”。

除了“标准版”之外，Sun还推出了两个其他的版本：一个是用于手机等嵌入式设备的“微型版”；另一个是用于服务器端处理的“企业版”。本书主要讲述标准版。

标准版的1.3和1.4版本对最初的Java 2版本做出了某些改进，扩展了标准类库，提高系统性能。当然，还修正了一些bug。在此期间，Java applet采用低调姿态，并淡化了客户端的应用，但却成为了服务器端应用的首选平台。

5.0版是自1.1版以来第一个对Java语言做出重大改进的版本（这一版本原来被命名为1.5版，在2004年的JavaOne会议之后，版本数字升至5.0）。经历了多年的研究，这个版本添加了泛型类型（generic type）（类似于C++的模版），其挑战性在于添加这一特性并没有对虚拟机做出任何修改。另外，还有几个来源于C# 的很有用的语言特性：for each循环、自动打包和元数据。语言的修改总会引起兼容性的问题。然而，这几个如此诱人的新语言特性，必将被程序设计人

员所接受。

版本6（没有后缀.0）于2006 年末发布。这个版本没有对语言方面再进行改进。但是，改进了其他性能，并增强了类库。表1-1展示了Java语言以及类库的发展状况。可以看到，应用程序接口（API）的规模发生了惊人的变化。

表1-1 Java语言的发展状况

版本	年份	语言新特性	类与接口的数量
1.0	1996	语言本身	211
1.1	1997	内部类	477
1.2	1998	无	1524
1.3	2000	无	1840
1.4	2004	断言	2723
5.0	2004	泛型类型、“for each”循环、可变元参数、自动打包、元数据、枚举、静态导入	3279
6	2006	无	3777

1.5 关于Java的常见误解

在结束本章之前，我们列出了一些关于Java的常见误解，同时给出了解释。

1）*Java是HTML的扩展。*

Java是一种程序设计语言；HTML是一种描述网页结构的方式。除了用于放置Java applet的HTML扩展之外，两者没有任何共同之处。

2）*使用XML，就不需要Java。*

Java是一种程序设计语言；XML是一种描述数据的方式。可以使用任何一种程序设计语言处理XML数据，而Java API对处理XML提供了很好的支持。此外，许多重要的第三方XML工具采用Java编写。有关这方面更加详细的信息请参看卷II。

3）*Java是一种非常容易学习的程序设计语言。*

像Java这种功能强大的语言大都不太容易学习。首先，必须将编写玩具式程序的轻松和开发实际项目的艰难区分开来。需要注意的是：本书只用了4章讨论Java语言。在两卷中，其他的章节介绍如何使用Java类库将Java语言应用到实际中去。Java类库包含了数千种类和接口与几万个方法。幸运的是，并不需要知道它们中的每一个，然而，要想Java解决实际问题，还是需要了解不少内容的。

4）*Java将成为适用于所有平台的通用性编程语言。*

从理论上讲，这是完全有可能的。的确，除了微软之外的每一个厂商都希望如此。然而，有很多在桌面计算机上已经工作良好的应用程序，在其他设备上或浏览器中或许不能正常地工作。同时，在编写这些应用程序时，利用了相应处理器的速度和本地的用户接口库，而且它们已经移植到所有重要的平台上。这类应用程序包括：字处理程序、图片编辑器以及Web浏览器。它们通常是用C或C++编写的，采用Java语言重新编写似乎对最终的用户不会带来什么特别的好处。

5）Java只不过是另外一种程序设计语言。

Java是一种很好的程序设计语言，很多程序设计人员喜欢Java胜过C、C++或C#。有上百种好的程序设计语言没有广泛地流行，而带有明显缺陷的语言，如：C++和Visual Basic却大行其道。

这是为什么呢？程序设计语言的成功更多地取决于其*支撑系统*的能力，而不是优美的语法。人们主要关注：是否提供了易于实现某些功能的易用、便捷和标准的库？是否拥有强大的程序设计能力与调试工具？语言和工具是否能够与计算机的其他基础结构整合在一起？Java的成功源于其类库能够让人们轻松地完成原本有一定难度的事情。例如：联网和多线程。Java减少了指针错误，因此使用Java编程的效率更高。但这些并不是Java成功的全部原因。

6）现在有了C#，Java过时了。

C#借鉴了Java许多好的思想，例如：清晰的语言结构、虚拟机和垃圾收集器。无论怎样，C#还是沿用了一些好的特性，其中最重要的是安全性和平台无关性。如果再能够与Windows捆绑在一起，就更加具有现实意义了。但是，从求职广告判定，Java仍然是大多数开发者选择的语言。

7）Java有专利，应该避免使用。

Sun Microsystems负责将Java的许可发放给销售者以及最终用户。尽管Sun公司通过Java Community Process最终控制着Java，但他们同时与许多其他的公司联手一起进行着语言修订版的开发及新类库的设计。虚拟机和类库的源代码都可以免费获取，但是，只能查阅，不能修改，也不能再发布。至此，Java已经“关闭源代码，但运转良好”。

这种状况在2007年发生了戏剧性的变化，Sun声称Java未来的版本将在General Public License下可用。Linux使用的是同一个开放源代码许可。要看到Sun管理Java未来的支配权还需要一段时间。然而，毋庸置疑，Java开放源代码已经算是一个非常勇敢的举动了，这会让Java的生存期延长很多年。

8）Java是解释型的，因此对于关键的应用程序速度太慢了。

早期的Java是解释型的。现在除了像手机这样的“微型”平台之外，Java虚拟机使用了即时编译器，因此采用Java编写的“热点”代码其运行速度与C++相差无几。

Java有一些C++没有的额外开销。虚拟机的启动时间要慢一些，并且Java GUI要比本地的GUI慢一些，这是因为它们采用了与平台无关的绘图方式。

对于Java比C++慢，人们已经抱怨很多年了。但是，今天的计算机速度远比人们发出抱怨的时候快了很多。一个较慢的Java程序与几年前相当快的C++程序相比还要快一些。就这一点来说，那些抱怨听起来有点像狐狸抱怨葡萄酸，有些人已经转过来攻击Java用户界面不够漂亮而不再攻击速度慢了。

9）所有的Java程序都是在网页中运行的。

所有的Java applet都是在网页浏览器中运行的。这也恰恰是applet的定义，即一种在网页中运行的Java程序。然而，大多数Java程序是运行在Web浏览器之外的独立应用程序。实际上，很多Java 程序都在Web服务器上运行并生成用于网页的代码。

本书的大多数程序都是独立的应用程序。诚然applet的确有趣，但是独立的Java程序在实际中更重要、更有用。

10）Java程序是主要的安全风险。

早期的Java，有过关于安全系统失效的报道，曾经一度引起公众哗然。大多数安全问题都存在于Java的特定浏览器中。研究人员将这视为一种挑战，即努力找出Java的漏洞，对applet安全模型的强度和复杂度发起挑战。随后，人们很快就解决了引发问题的所有技术因素。据我们所知，任何实用系统都有安全危机。想想看：毫不夸张地说，有数百万种病毒攻击着Windows的可执行文件和Word宏，这给系统造成了巨大的损害，但却很少有人批评被攻击平台的脆弱。同样，Internet Explorer中的ActiveX机制始终作为被攻击的目标，但由于阻止这种攻击非常简单，所以人们也就懒得将它们公布于众了。

有些系统程序员在公司的浏览器中禁用Java，却允许其用户下载可执行文件、ActiveX 控件和Word文档。这是多么荒唐可笑啊！通常，不友好的Java applet的袭击的风险大约相当于因飞机失事遇难的风险；而打开Word文档遭到袭击的风险则相当于步行横穿繁忙的高速公路遇难的风险。

11）JavaScript是Java的简易版。

JavaScript是一种在网页中使用的脚本语言，它是由Netscape 发明的，原来的名字叫做LiveScript。JavaScript 的语法类似Java，除此之外，两者无任何关系。当然，名字有些相像。JavaScript的一个子集已经标准化为ECMA-262。与Java applet相比，JavaScript更紧密地与浏览器集成在一起。特别是JavaScript 程序可以修改正在显示的文档，而applet只能在有限的区域内控制外观。

12）使用Java可以用价值500美元的Internet设备取代电脑。

当Java刚刚发布的时候，一些人打赌：肯定会有这样的好事情发生。从本书的第1版开始，我们就已经认定“家庭用户将会放弃功能强大且便利的桌面系统，而使用没有本地存储的网络设备”是一种荒谬的想法。我们发现基于Java的网络计算机，对利用“零管理”降低计算机所有者的商业成本是一种很好的选择。即便如此，这种好事也没有发生。

另一方面，Java已经广泛地用在手机上。我们必须承认还没有看到一个运行在手机上的Java应用程序是必不可少的。但是，常见的游戏和屏幕保护程序在许多市场上销售得很好。

! 提示：有关Java常见问题的解答可以通过Java FAQ（Java Frequently Question）列出的网站查找：http://www.apl.jhu.edu/~hall/java/FAQs-and-Tutorials.html。

第2章 Java程序设计环境

▲ 安装Java 开发工具箱
▲ 选择开发环境
▲ 使用命令行工具
▲ 使用集成开发环境
▲ 运行图形化应用程序
▲ 建立并运行applet

本章主要介绍如何安装Java开发工具箱（JDK）以及如何编译和运行各种类型的程序：控制台程序、图形化应用程序以及applet应用程序。运行JDK的方法是在shell窗口中键入命令行。然而，很多程序员更喜欢使用集成开发环境。为此，将在稍后介绍如何使用免费的开发环境编译和运行Java程序。尽管学起来很容易，但集成开发环境需要吞噬大量资源，在编写小型程序时会给人带来烦恼。作为折中方案，再介绍一下如何调用Java编译器并运行Java程序的文本编辑器。一旦掌握了本章的技术，并选定了自己的开发工具，就可以学习第3章，开始研究Java程序设计语言。

2.1 安装Java开发工具箱

Sun Microsystems公司为Solaris、Linux和Windows提供了Java开发工具箱（JDK）的最新、最完整的版本。用于Macintosh和一些其他平台的版本仍处于各种不同的开发状态中，不过，这些版本都是由相应平台的开发商授权并分发的。

注释：有些Linux的发行版已经预先打包了JDK。例如，在Ubuntu中，可以用apt-get或Synaptic GUI安装sun-java6-jdk包来安装JDK。

2.1.1 下载JDK

要想下载Java开发程序包，必须到Sun网站进行搜寻，并且在得到所需的软件之前必须弄清大量的专业术语。请看表2-1。

表2-1 Java术语

术语名	缩写	解释
Java Development Kit	JDK	编写Java程序的程序员使用的软件
Java Runtime Environment	JRE	运行Java程序的用户使用的软件
Standard Edition	SE	用于桌面或简单的服务器应用的Java平台
Enterprise Edition	EE	用于复杂的服务器应用的Java平台
Micro Edition	ME	用于微型手机cell phone和其他小型设备的Java平台
Java 2	J2	一个过时的术语，用于描述1998年~2006年之间的Java版本
Software Development Kit	SDK	一个过时的术语，用于描述1998年~2006年之间的JDK
Update	u	Sun的术语，用于发布修改的bug
NetBeans	—	Sun的集成开发环境

JDK是Java Development Kit的缩写。有点混乱的地方是：工具箱的版本1.2~版本1.4被称为Java SDK（Software Development Kit）。在某些场合下，还可以看到这些过时的术语。另外，还有Java 运行时环境（JRE），它包含虚拟机但不包含编译器。这并不是开发者所想要的环境，而是专门为不需要编译器的用户而设计。

还有，随处可见的Java SE，相对于Java EE（Enterprise Edition）和Java ME（Micro Edition），它是Java的标准版。

Java 2这种提法始于1998年。当时Sun公司的销售人员感觉通过增加小数点后面的数值改变版本号并没有反映出JDK1.2 的重大改进。但是，由于在发布之后才意识到这个问题，所以决定将开发工具箱的版本号仍然沿用1.2，接下来的版本就是1.3、1.4和5.0。但是，Java平台被重新命名为Java 2。因此，就有了Java 2 Standard Edition Software Development Kit 的5.0版，即J2SE SDK 5.0。

对于工程师来说，所有这一切都可能会引起困惑，这正是没有将其投入市场的原因。2006年，日趋完善的Java SE开始流行。无意义的Java 2被遗弃，Java当前的标准版本被称为Java SE 6。偶尔还会看到使用1.5版本和1.6版本，但这些只是5.0版本和6版本的同义词。

最后，当Sun为解决一些紧急问题做出了某些微小的版本改变时，将其称为更新。例如：对Java SE 6开发包做出的第一次更新，正式称为JDK 6u1，内部的版本号为1.6.0_01。

如果使用Solaris、Linux或Windows，就应该从http://java.sun.com/javase下载Java软件开发工具箱。找到版本6或后续的版本，并选择自己的平台。如果软件被称为“更新”也不必担心。更新将捆绑包含了整个JDK的最新版本。

Sun公司曾经制作过将Java开发工具箱和集成开发环境捆绑在一起的产品。其中的集成开发环境，在不同时期，被命名为不同的名字，例如，Forte、Sun ONE Studio、Sun Java Studio和NetBeans。我们无法知道每个人在登录Sun网站时，市场正在热销什么。这里，建议大家只安装Java开发工具箱。如果最终决定使用Sun的集成开发环境，就可以从http://netbeans.org下载。

下载JDK之后，随后的安装过程请参看不同平台的安装指南。在编写本书时，可以在网站http://java.sun.com/javase/6/webnotes/install/index.html上搜寻到。

只有Java的安装过程和编译命令与系统有关。一旦安装并运行了Java，本书阐述的所有内容都适用。与系统无关性是Java的一个主要优势。

注释：安装过程提供了包含JDK版本号（如jdk1.6.0）的默认的安装路径。这似乎有些烦人，但是，应该重视版本号，它会给安装新版JDK的测试带来便利。

在Windows环境下，强烈建议不要接受带空格的默认路径名，如：c:\ProgromFiles\jdk1.6.0。应该将Progrom Files部分删掉。

在本书中，使用的安装路径是jdk。例如：当引用jdk/bin目录时，意味着引用的是/usr/local/jdk1.6.0/bin或c:\ jdk1.6.0\bin。

2.1.2 设置执行路径

在完成了JDK的安装之后，还需要执行另外一个步骤：把jdk/bin目录添加到执行路径中。所谓执行路径是指操作系统搜索本地可执行文件的目录列表。对于不同的操作系统，这个步骤

的操作过程有所不同。

- 在UNIX（包括Solaris和Linux）环境下，编辑执行路径的过程与所使用的shell有关。如果使用的是C shell（Solaris的默认选择），在 ~/.cshrc文件的末尾就要添加：

```
set path=(/usr/local/jdk/bin $path)
```

如果使用的是Bourne Again shell（Linux的默认选择），在 ~/.bashrc 文件或~/.bash_profile文件的末尾就要添加：

```
export PATH=/usr/local/jdk/bin:$PATH
```

- 在Windows下，以管理员身份登录。启动Control Panel，切换到Classic View，并选择System图标。在Windows NT/2000/XP中，立即会看到System Properties对话框。在Vista中，需要选择Advanced系统设置（如图2-1所示）。在System Properties对话框中，点击Advanced标签，然后点击Environment Variables按钮。滚动System Variables窗口直到找到变量名path为止。点击Edit按钮（如图2-2所示）。将jdk\bin目录添加到路径的开始处，用分号将新条目隔开，如下所示：

```
c:\jdk\bin;other stuff
```

将这个设置保存起来，打开的任何控制窗口都会有正确的路径。

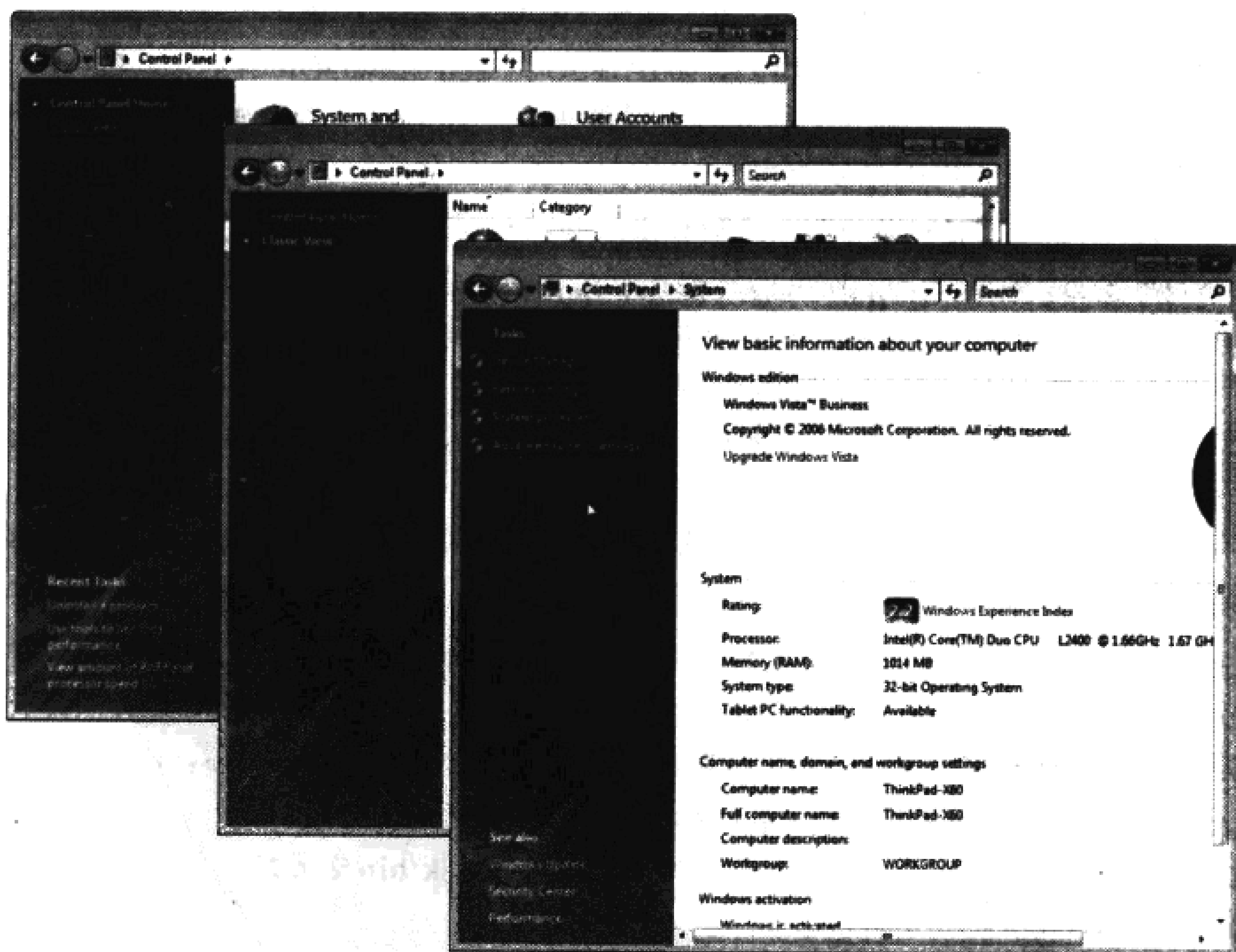


图2-1 在Windows Vista中启动的System Properties对话框

下面是测试上述设置是否正确的方法：打开一个shell窗口，键入：

```
java -version
```

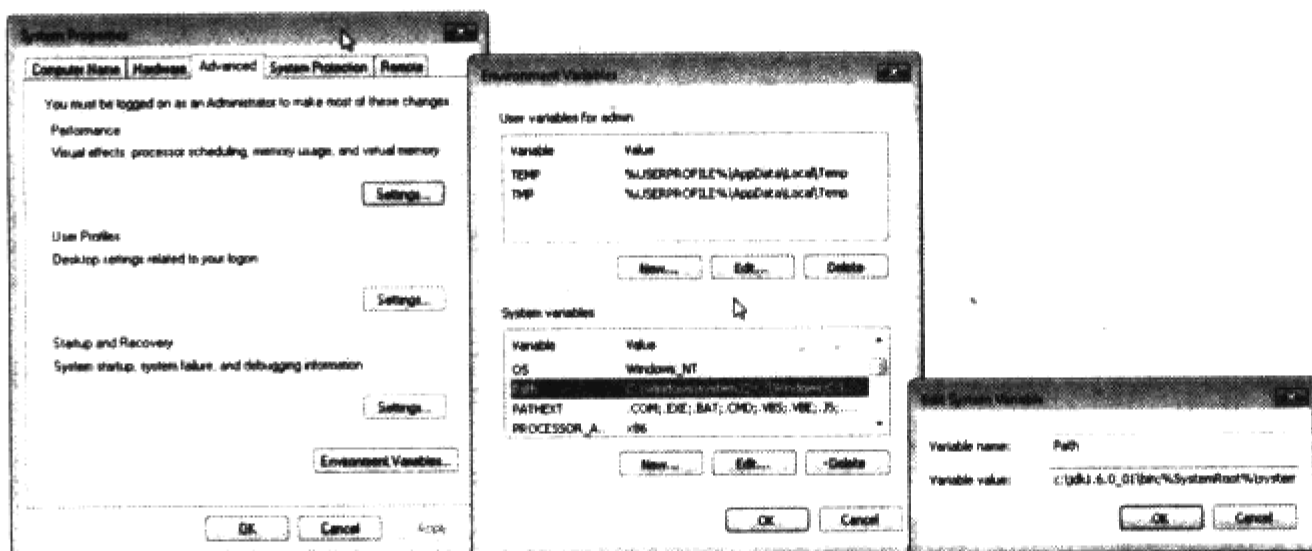


图2-2 在Windows Vista中设置Path环境变量

然后，按ENTER键。应该能够看到下面的显示信息：

```
java version "1.6.0_01"
Java(TM) SE Runtime Environment (build 1.6.0_01-b06)
Java HotSpot(TM) Client VM (build 1.6.0_01-b06, mixed mode, sharing)
```

如果看到的是"java:commond not found"、或"The name specified is not recogonized as an internel or external commond, operable program or batch file"，则需要回到前面，重新检查整个的安装过程。

> **注释**：在Windows中，按照下面的命令打开shell窗口。如果使用Windows NT/2000/XP环境，那么在开始菜单中选择Run选项，并键入cmd。在Vista中，只在开始菜单中的Start Search中输入cmd，再按下ENTER键就会出现一个shell窗口。
>
> 如果曾经没有接触过这些内容，建议学习一下有关命令行的基础教程。很多大学的计算机科学系在网上都有相关的教程，例如http://www.cs.sjsu.edu/facult/horstman/CS46A/windows/tutorial. html。

2.1.3 安装源代码库和文档

库源文件在JDK中以一个压缩文件src.zip 的形式发布，必须将其解压缩后才能够访问源代码。这里强烈地建议按照下面所述的步骤进行操作。很简单：

1）确保JDK已经安装，并且jdk/bin目录在执行路径中。

2）打开shell窗口。

3）进入jdk目录（例如：cd /usr/local/jdk1.6.0 或 cd c:\jdk1.6.0）。

4）建立一个子目录src

```
mkdir src
cd src
```

5）执行命令：

```
jar xvf ../src.zip
```

（或者在Windows中执行jar xvf ..\src.zip。）

提示：在src.zip文件中包含了所有公共类库的源代码。要想获得更多的源代码（例如：编译器、虚拟机、本地方法以及私有辅助类），请访问网站：http://download.java.net/jdk6。

文档包含在一个压缩文件中，它是一个独立于JDK的压缩文件。可以直接从网站http://java.sun.com/javase/downloads下载获得这个文档。操作步骤如下：

1）确认JDK已经安装，并且jdk/bin目录在执行路径上。

2）下载文档压缩文件并将其存放在jdk目录下。这个文件名为jdk-*version*-doc.zip，其中的*version*表示版本号，例如，6。

3）打开一个shell窗口。

4）进入jdk目录。

5）执行命令：

```
jar xvf jdk-version-doc.zip
```

其中*version*是相应的版本号。

2.1.4 安装本书中的示例

读者可以安装本书中的示例程序。这些程序可以从http://horstmann.com/corejava下载，它们都打包在corejava.zip文件中。应该将它们解压到一个单独的文件夹中，建议将文件夹命名为CoreJavaBook。需要执行下列步骤：

1）确保JDK已经安装，并且jdk/bin目录在执行路径中。

2）建立目录CoreJavaBook。

3）将corejava.zip下载到这个目录下。

4）打开一个shell窗口。

5）进入CoreJavaBook目录。

6）执行命令：

```
jar xvf corejava.zip
```

2.1.5 导航Java目录

在学习Java的过程中，经常需要查看Java源文件。当然，也会频繁地使用类库文档。表2-2显示了JDK目录树。

表2-2 JDK目录树

目录结构	描　述
jdk	（名字可能不同，例如：jdk5.0）
├──*bin*	编译器和工具
├──*demo*	演示
├──*docs*	HTML格式的类库文档（展开j2sdkversion-doc.zip之后）
├──*include*	用于编译本地方法的文件（参看卷II）
├──*jre*	Java运行环境文件
├──*lib*	类库文件
└──*src*	类库源文件（展开src.zip之后）

就学习Java而言，docs和src是两个最有用的子目录。docs目录包含了HTML格式的类库文档，可以使用任何浏览器（如：Netscape）查看这些文档。

! **提示**：在浏览器中设置一个指向docs/api/index.html书签。使用Java平台时经常需要查看这一页的内容。

src目录包含了Java类库中公共部分的源代码。当对Java熟悉到一定程度时，可能会感到本书以及联机文档已经无法提供所需的信息了。那时，应该深入研究Java的源代码。请放心，只要深入地研究源代码就一定会搞清楚类库函数的真正功能。例如，如果对System类的内部工作感到好奇，可以查看src/java/lang/System.java。

2.2 选择开发环境

如果在此之前使用Microsoft Visual Studio 编写过程序，那么就会习惯于带有内嵌的文本编辑器、用于编译和运行程序的菜单，以及配有集成调试器的开发环境。基本的JDK全然没有这些功能。利用它完成每一项任务时都要在shell窗口中键入命令。这些听起来很麻烦，但只是一个基本的技能。第一次安装Java时，可能希望在安装开发环境之前检测一下安装的Java是否正确。此外，还可以通过执行一些基本的操作步骤，加深对开发环境幕后工作的理解。

然而，在掌握了编译和运行Java程序的基本步骤之后，你就想要使用专业的开发环境了。在过去的10年中，这些环境变得功能非常强大，操作也非常方便，以至于不选用它们将是极其不理智的。使用Eclipse和NetBeans这两个免费的开发环境是一个很好的选择。尽管NetBeans 发展的速度比较快，但在本章中，还是打算介绍如何使用Eclipse，这是因为Eclipse要比NetBeans更加灵活一些。当然，如果青睐使用其他的开发环境，同样也可以完成本书的所有程序。

在过去，推荐使用文本编辑器编写简单的程序，如：Emacs、JEdit或者TextPad。现在不会再这样推荐了，因为集成开发环境非常快捷、方便。

总之，应当了解如何使用基本的JDK工具，这样才会感觉使用集成开发环境是一种享受。

2.3 使用命令行工具

首先介绍较难的一种方法：通过命令行编译并运行Java程序。

1）打开一个shell窗口。

2）进入CoreJavaBook/v1ch02/Welcome目录（CoreJavaBook是安装本书示例源代码的目录，请参看“安装本书中的示例”一节）。

3）键入下面的命令：

```
javac Welcome.java
java Welcome
```

然后，将会在shell窗口中看到图2-3所示的输出。

祝贺你！已经编译并运行了第一个Java程序。

那么，刚才都进行了哪些操作呢？javac程序是一个Java编译器。它将文件Welcom.java编译成Welcom.class，并发送到Java虚拟机。虚拟机执行编译器存放在class文件中的字节码。

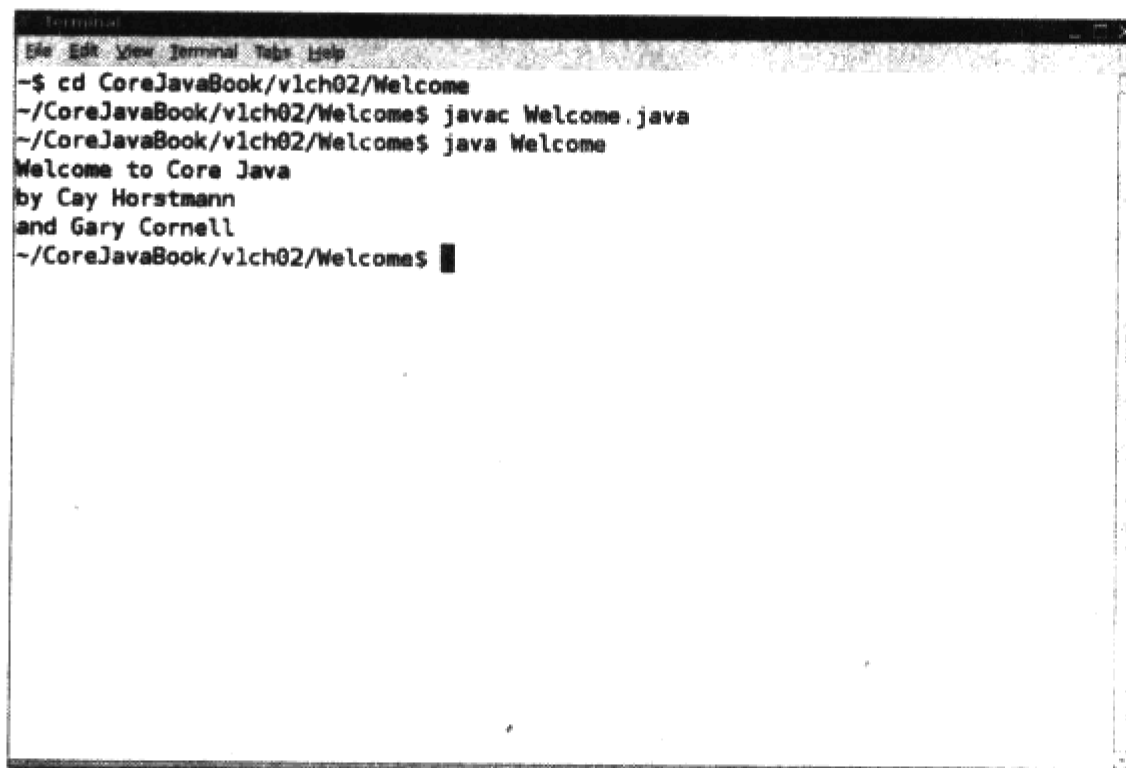


图2-3 编译并运行Welcom.java

注释：如果得到了下面这行语句的错误消息

```
for (String g : greeting)
```

就可能是使用了Java编译器的旧版本。Java SE 5.0为Java程序设计语言增加了一些非常令人欢快的新特性，在本书中，使用了这些特性。

如果使用的是Java旧版本，则需要将上面的循环语句改写成下列形式：

```
for (int i = 0; i < greeting.length; i++)
   System.out.println(greeting[i]);
```

Welcome程序非常简单。其功能只是向控制台输出了一条消息。看到例2-1的程序代码可能非常满意。(在下一章中，将解释它是如何工作的。)

例2-1 Welcome.java

```
/**
 * This program displays a greeting from the authors.
 * @version 1.20 2004-02-28
 * @author Cay Horstmann
 */
public class Welcome
{
   public static void main(String[] args)
   {
      String[] greeting = new String[3];
      greeting[0] = "Welcome to Core Java";
      greeting[1] = "by Cay Horstmann";
      greeting[2] = "and Gary Cornell";

      for (String g : greeting)
         System.out.println(g);
   }
}
```

疑难解答提示

在使用可视化开发环境的年代，许多程序员对于在shell窗口中运行程序已经很生疏了。常常会出错很多错误，最终导致令人沮丧的结果。

一定要注意以下几点：

- 如果手工地输入源程序，一定要注意大小写。尤其是类名为Welcome，而不是welcome或WELCOME。
- 编译时需要提供一个*文件名*（Welcome.java），而运行时，只需要指定*类名*（Welcome），不要带扩展名.java或.class。
- 如果看到诸如Bad command or file name或javac:command not found这类消息，就要返回去检查一下安装是否出现了问题，特别是执行路径的设置。
- 如果javac报告了一个错误"cannot read：Welcome.java"，就应该检查一下目录中是否存在这个文件。

在UNIX环境下，检查Welcome.java是否以正确的大写字母开头。在Windows环境下，使用shell命令dir，而不要使用图形浏览器工具。有些文本编辑器（特别是Notepad）在每个文件名后面要添加扩展名.txt。如果使用Notepad编辑Welcome.java 就会存为Welcome.java.txt。对于默认的Windows设置，浏览器与Notepad都隐含.txt的扩展名，这是因为这个扩展名属于"已知的文件类型"。此时，需要重新命名这个文件，使用shell命令ren，或是另存一次，给文件名加一对双引号，如："Welcome.java"。

- 如果运行程序之后，收到错误消息java.lang.NoClassDefFoundError，就应该仔细地检查一下类名。

如果收到关于welcome（w为小写）的错误消息，就应该重新发出带有大写W的命令：java Welcome。记住，Java区分大小写。

如果收到有关Welcome/java的错误信息，就是错误地键入了java Welcome.java，应该重新输入命令java Welcome。

- 如果键入java Welcome，而虚拟机没有找到Welcome类，就应该检查一下是否系统的CLASSPATH环境变量被别人更改了（将这个变量设置为全局并不是一个提倡的做法，然而，Windows中有些编写很差的软件安装程序就是这样做的）。可以在当前的shell窗口中键入下列命令，临时地取消CLASSPATH环境变量的设置：

```
set CLASSPATH=
```

这个命令应用于使用C shell的 Windows和UNIX/Linux环境下。在使用Bourne/bash shell 的UNIX/Linux环境下需要使用：

```
export CLASSPATH=
```

- 如果收到的错误信息是有关新语言结构的问题，就需要确认一下当前使用的编译器是否支持Java SE 5.0。
- 如果程序中的错误太多，所有的错误消息就会飞快地闪过。编译器将会把这些错误消息发送到标准错误流上。如果消息超过一屏，就很难看清楚它们了。可以使用shell的操作符2>将这些错误重定向到一个文件中：

```
javac MyProg.java 2> errors.txt
```

提示： 在http://java.sun.com/docs/books/tutorial/getStarted/cupojava/上有一个很好的教程。其中提到了一些初学者经常容易犯的错误。

2.4 使用集成开发环境

本节将介绍如何使用Eclipse编译一个程序。Eclipse是一个可以从网站http://eclipse.org上免费下载得到的集成开发环境。Eclipse是采用java编写的，然而，由于所使用的是非标准视窗类库，所以，不如Java那样具有可移植性。尽管如此，这个开发环境已经拥有可以应用在Linux、Mac OS X、Solaris以及Windows的版本了。

还有一些使用比较普遍的IDE，但就目前而言，Eclipse是最通用的。使用步骤如下所示：

1）启动Eclipse之后，从菜单选择File→New Project。

2）从向导对话框中选择Java Project（如图2-4所示）。这些是Eclipse 3.2的屏幕画面。如果使用的Eclipse版本稍有差异，不必介意。

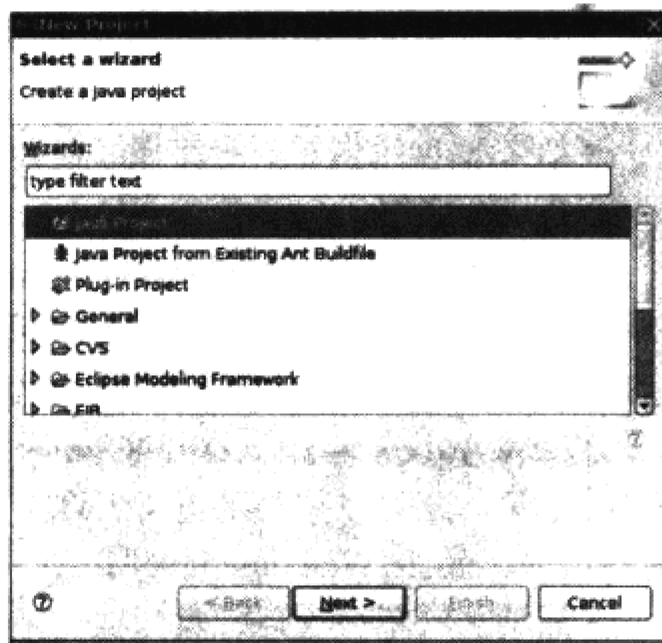


图2-4 Eclipse中的“New Project”对话框

3）点击Next按钮，键入工程名Welcome，并输入包含Welcome.java的完整路径名。如图2-5所示。

4）确保没有选定标有Create project in workspace的选项。

5）点击Finish按钮。这个工程已经创建完成了。

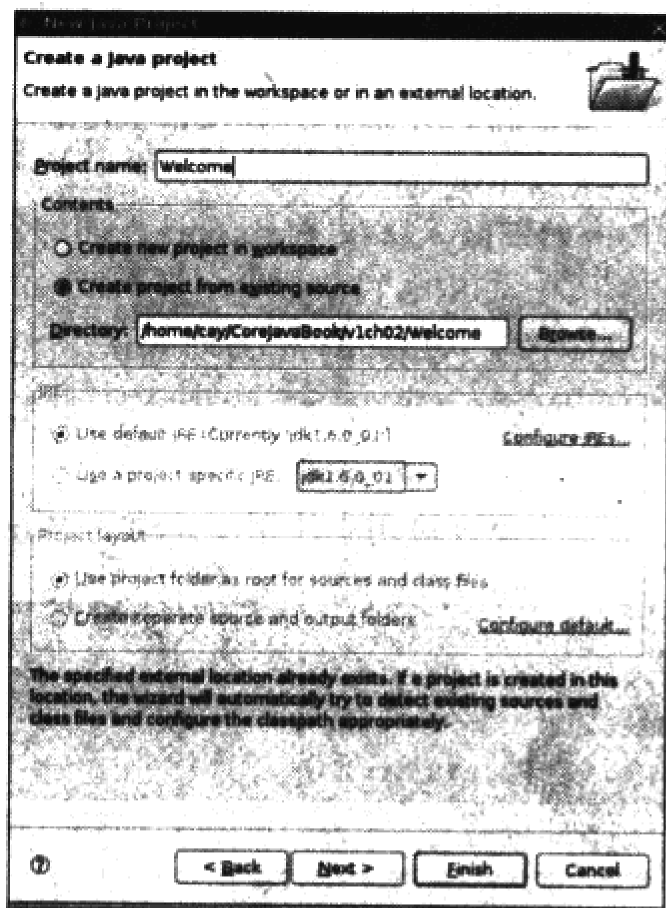


图2-5 配置Eclipse工程

6）点击在工程窗口左边窗格中的三角，并打开它。然后，点击旁边的Default package三角，再双击Welcome.java。现在应该看到带有程序代码的窗口了（如图2-6所示）。

7）用鼠标右键点击最左侧窗格中的工程名（Welcome），选择Run→Run As→Java Application。可以看到：这个窗口底部出现了一个输出窗口。在这个输出窗口中显示了程序的输出结构（如图2-7所示）。

定位编译错误

可以肯定，这个程序没有出现输入错误或bug（毕竟，这段代码只有几行）。为了说明问题，假定在代码中不小心出现了录入错误（或许语法错误）。试着将原来的程序修改一下，让它包含一些录入错误，例如，将String的大小写弄错：

```
public static void main(string[] args)
```

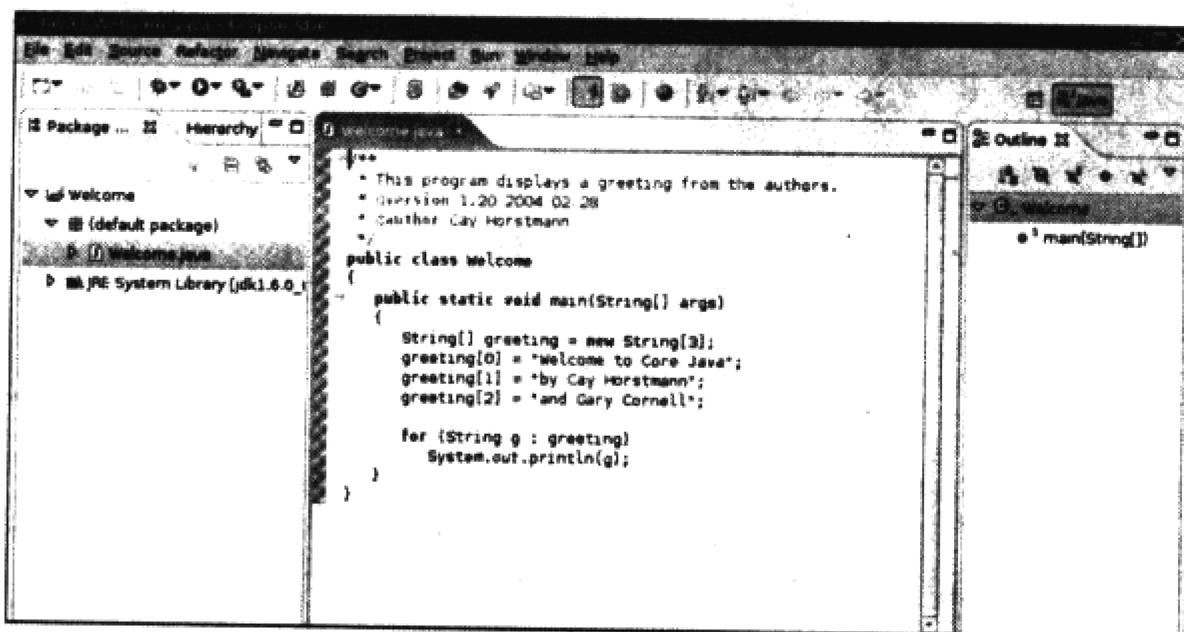


图2-6 使用Eclipse编辑源文件

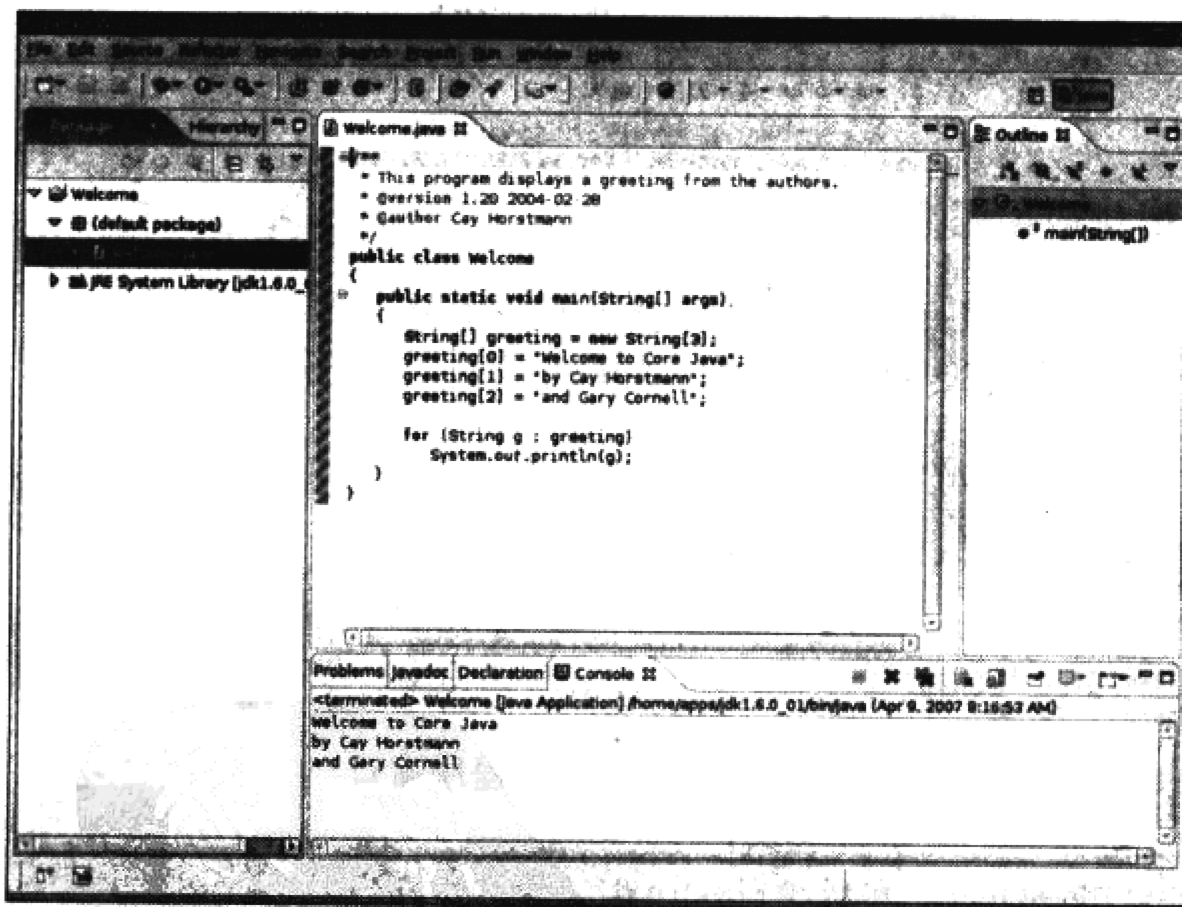


图2-7 在Eclipse 中运行程序

现在，重新编译这个程序，就会收到一条错误消息，其中指出了有一个不能识别的string类型（如图2-8所示）。点击一下错误消息，可以看到光标马上移到了编辑窗口中相应的行上，在这里将错误纠正过来。使用这种方式可以快速地纠正错误。

! **提示**：通常，Eclipse错误报告会伴有一个加亮的图标。点击这个图标可以得到一个建议解决这个错误的方案列表。

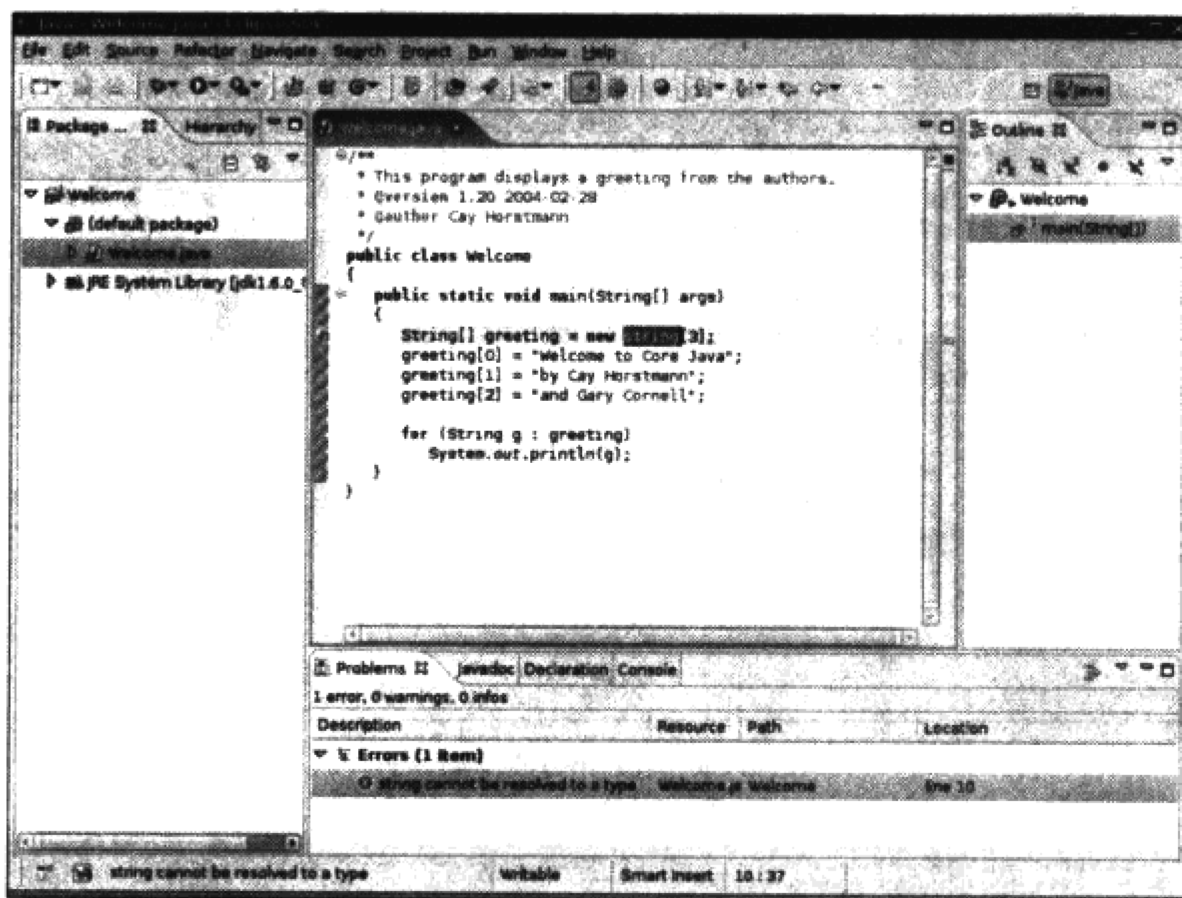


图2-8 Eclipse的错误消息

2.5 运行图形化应用程序

Welcome程序并不会引起人们的兴奋。接下来，给出一个图形化应用程序。这个程序是一个简单的图像文件查看器（viewer），它可以加载并显示一个图像。首先，由命令行编译并运行这个程序。

1）打开一个shell窗口。

2）进入CoreJavaBook/v1ch02/ImageViewer。

3）输入：

```
javac ImageViewer.java
java ImageViewer
```

运行后将弹出一个标题栏为ImageViewer的新程序窗口（如图2-9所示）。

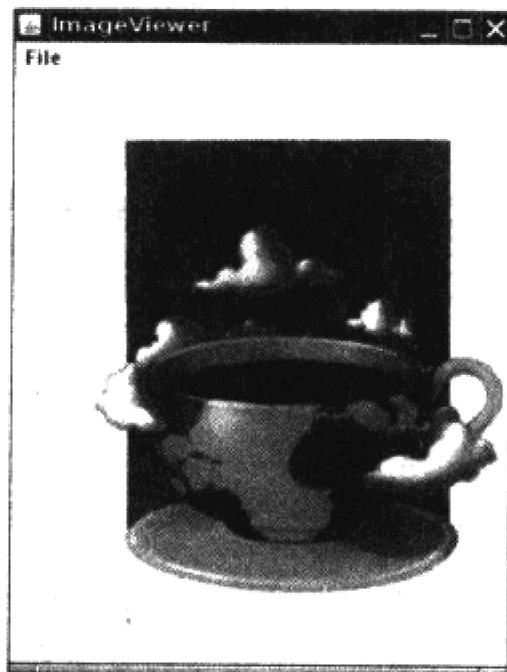


图2-9 运行ImageViewer应用程序

现在，选择File→Open，然后找到一个GIF文件并打开它（在同一个目录下提供了两个示例文件）。

要想关闭这一程序，只需要点击标题栏中的关闭按钮或者从系统菜单中选择关闭程序（要在文本编辑器中或集成开发环境中编译并运行程序，按照前面的步骤做就行了。例如，在Emacs中选择JDE→Compile，然后，再选择JDE→Run App）。

现在读者一定会感觉这个程序既有趣又有用。下面快速地浏览一下源代码。这个程序比第一个程序要长很多，但是只要想一想用C 或C++编写同样功能的应用程序所需要的代码

量，就不会感到它太复杂了。当然，在Visual Basic中，编写这个程序相当简单，只需要简单的拖放几下，再加上两行代码就行了。JDK没有可视化的界面构造器，所以必须通过编写代码完成这一切工作，如例2-2所示。本书将在第7章~第9章介绍编写图形化应用程序的内容。

例2-2 ImageViewer.java

```
import java.awt.EventQueue;
import java.awt.event.*;
import java.io.*;
import javax.swing.*;

/**
 * A program for viewing images.
 * @version 1.22 2007-05-21
 * @author Cay Horstmann
 */
public class ImageViewer
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new ImageViewerFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * A frame with a label to show an image.
 */
class ImageViewerFrame extends JFrame
{
   public ImageViewerFrame()
   {
      setTitle("ImageViewer");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // use a label to display the images
      label = new JLabel();
      add(label);

      // set up the file chooser
      chooser = new JFileChooser();
      chooser.setCurrentDirectory(new File("."));

      // set up the menu bar
      JMenuBar menuBar = new JMenuBar();
      setJMenuBar(menuBar);

      JMenu menu = new JMenu("File");
```

```
      menuBar.add(menu);

      JMenuItem openItem = new JMenuItem("Open");
      menu.add(openItem);
      openItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               // show file chooser dialog
               int result = chooser.showOpenDialog(null);

               // if file selected, set it as icon of the label
               if (result == JFileChooser.APPROVE_OPTION)
               {
                  String name = chooser.getSelectedFile().getPath();
                  label.setIcon(new ImageIcon(name));
               }
            }
         });
      JMenuItem exitItem = new JMenuItem("Exit");
      menu.add(exitItem);
      exitItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               System.exit(0);
            }
         });
   }

   private JLabel label;
   private JFileChooser chooser;
   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 400;
}
```

2.6 建立并运行applet

本书给出的前两个程序是Java应用程序。它们与所有本地程序一样，可以独立地运行。然而，正如第1章提到的，有关Java的大量宣传都在炫耀Java能够在浏览器中运行applet的能力。下面介绍一下如何利用命令行建立并运行applet。然后，利用JDK自带的applet查看器加载applet。最后，在Web浏览器中显示。

首先，打开shell窗口并将目录转到CoreJavaBook/v1ch02/WelcomeApplet，然后，输入下面的命令：

```
javac WelcomeApplet.java
appletviewer WelcomeApplet.html
```

图2-10显示了在applet查看器窗口中显示的内容。

第一条命令是大家已经非常熟悉的调用Java编译器的命令。它将WelcomeApplet.java源文件编译成字节码文件WelcomeApplet.class。

在这里，不要运行Java程序，而调用appletviewer程序。这是JDK自带的一个特殊工具，它

可以帮助人们快速地测试applet。这里需要向这个程序提供一个HTML文件，而不是Java类文件名。WelcomeApplet.html的内容请参看下面的例2-3。

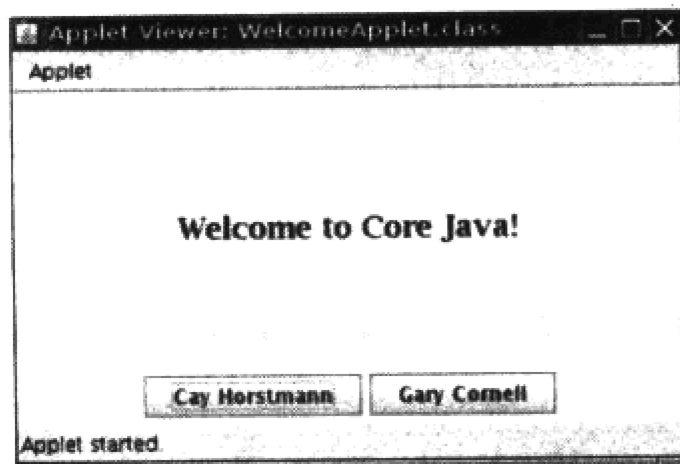


图2-10 在applet查看器窗口中显示的WelcomeApplet

例2-3 WelcomeApplet.html

```
<html>
   <head>
      <title>WelcomeApplet</title>
   </head>
   <body>
      <hr/>
      <p>
         This applet is from the book
         <a href="http://www.horstmann.com/corejava.html">Core Java</a>
         by <em>Cay Horstmann</em> and <em>Gary Cornell</em>,
         published by Sun Microsystems Press.
      </p>
      <applet code="WelcomeApplet.class" width="400" height="200">
         <param name="greeting" value ="Welcome to Core Java!"/>
      </applet>
      <hr/>
      <p><a href="WelcomeApplet.java">The source.</a></p>
   </body>
</html>
```

如果熟悉HTML就会注意到一些标准的HTML指令和applet标记，它们用于告诉applet查看器需要加载一个applet，其代码存放于WelcomeApplet.class中。applet查看器将忽略applet标记之外的所有HTML标记。遗憾的是，浏览器的情况有点复杂。

- 在Windows、Linux 和Mac OS X环境下，Firefox支持Java 。要想实验一下applet，只需要下载这些浏览器的最新版本（访问http://java.com），并利用版本检查器查看一下是否需要安装Java插件。
- Internet Explorer的有些版本根本不支持Java。其他的也只支持Microsoft Java Virtual Machine的过时版本。如果运行Internet Explorer，就需要从http://java.com上下载并安装Java插件。
- 如果使用Macintosh运行OS X，Safari与 Macintosh的Java被集成在一起。在编写本书时支

持Java SE 5.0。

假定有一个支持目前流行的Java版本的浏览器，可以在浏览器中试着加载applet。

1）启动浏览器。

2）选择File→Open File (或等效的操作)。

3）进入CoreJavaBook/v1ch02/WelcomeApplet目录。在文件对话框中，将会看到WelcomeApplet.html文件。加载这个文件。

4）浏览器将加载applet，并显示文字。显示效果基本上如图2-11所示。

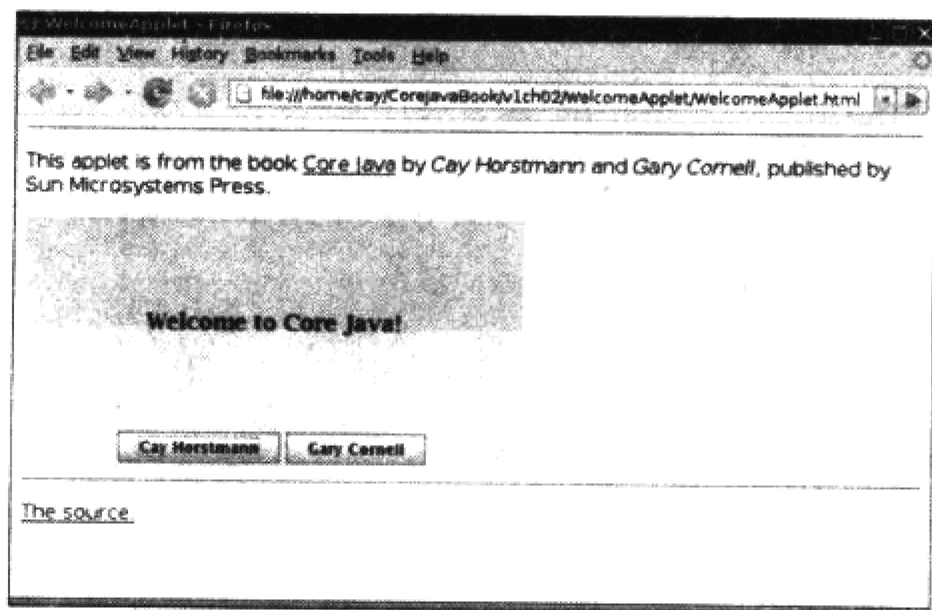


图2-11 在浏览器中运行WelcomeApplet

读者可能会发现这个应用程序是活动的，并且可以与Internet进行交互。点击Cay Horstmann按钮，applet会让浏览器显示Cay的网页。点击Gray Cornell 按钮，applet会让浏览器弹出一个邮件窗口，其中包含已经填写好的Gray邮件地址。

需要注意的是，在查看器中的两个按钮并不起作用。applet查看器没有能力发送邮件或显示一个网页，因此会忽略这里的操作请求。applet查看器适于用来单独地测试applet，但是，最终需要将applet放到浏览器中，以便检测与浏览器以及Internet的交互情况。

提示：*也可以从编辑器或集成开发环境内部运行applet。在Emac中，从菜单中选择JDE→Run Applet。在Eclipse中，使用Run→Run as→Java Applet菜单选项。*

最后，在例2-4中给出了这个applet的代码。现在，只要阅读一下就可以了。在第10章中，还会再次介绍applet的编写。

例2-4 WelcomeApplet.java

```
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import javax.swing.*;

/**
 * This applet displays a greeting from the authors.
 * @version 1.22 2007-04-08
```

```
 * @author Cay Horstmann
 */
public class WelcomeApplet extends JApplet
{
   public void init()
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               setLayout(new BorderLayout());

               JLabel label = new JLabel(getParameter("greeting"), SwingConstants.CENTER);
               label.setFont(new Font("Serif", Font.BOLD, 18));
               add(label, BorderLayout.CENTER);

               JPanel panel = new JPanel();

               JButton cayButton = new JButton("Cay Horstmann");
               cayButton.addActionListener(makeAction("http://www.horstmann.com"));
               panel.add(cayButton);

               JButton garyButton = new JButton("Gary Cornell");
               garyButton.addActionListener(makeAction("mailto:gary_cornell@apress.com"));
               panel.add(garyButton);

               add(panel, BorderLayout.SOUTH);
            }
         });
   }

   private ActionListener makeAction(final String urlString)
   {
      return new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               try
               {
                  getAppletContext().showDocument(new URL(urlString));
               }
               catch (MalformedURLException e)
               {
                  e.printStackTrace();
               }
            }
         };
   }
}
```

在本章中，学习了有关编译和运行Java程序的机制。现在可以转到第3章开始学习Java语言了。

第3章　Java基本的程序设计结构

- ▲ 一个简单的Java应用程序
- ▲ 注释
- ▲ 数据类型
- ▲ 变量
- ▲ 运算符
- ▲ 字符串
- ▲ 输入输出
- ▲ 控制流程
- ▲ 大数值
- ▲ 数组

现在，假定已经成功地安装了JDK，并且能够运行第2章中给出的示例程序。我们从现在开始将介绍Java应用程序设计。本章主要讲述程序设计相关的基本概念（如数据类型、分支以及循环）在Java中的实现方式。

非常遗憾，需要告诫大家，使用Java编写GUI应用程序并不是一件很容易的事情，编程者需要掌握很多相关的知识才能够创建窗口、添加文本框和按钮等。介绍基于GUI的Java应用程序设计技术与本章将要介绍的程序设计基本概念相差甚远，因此本章给出的所有示例都是为了说明一些相关概念而设计的“玩具式”程序，它们仅仅通过shell窗口输入输出。

最后需要说明，对于一个具有C++编程经验的程序员来说，本章的内容只需要浏览一下，应该重点阅读分布在正文中的C/C++注释。对于具有使用Visual Basic等其他编程背景的程序员来说，可能会发现其中的绝大多数概念都很熟悉，但是在语法上有比较大的差异，因此，需要非常仔细地阅读本章的内容。

3.1　一个简单的Java应用程序

下面看一个最简单的Java应用程序，它只发送一条消息到控制台窗口中：

```
public class FirstSample
{
   public static void main(String[] args)
   {
      System.out.println("We will not use 'Hello, World!'");
   }
}
```

这个程序虽然很简单，但所有的Java应用程序都具有这种结构，还是值得花一些时间研究一下。首先，Java对大小写敏感。如果出现了大小写拼写错误（例如，将main拼写成Main），那程序将无法运行。

下面逐行地查看一下这段源代码。关键字public称为访问修饰符（access modifier），它用于控制程序的其他部分对这段代码的访问级别。在第5章中将会更加详细地介绍访问修饰符的具体内容。关键字class表明Java程序中的全部内容都包含在类中。这里，只需要将类作为一个加载程序逻辑的容器，程序逻辑定义了应用程序的行为。在第4章中将会用大量的篇幅介绍

Java类。正如第1章所述，类是构建所有Java应用程序和applet的构建块。Java应用程序中的全部内容都必须放置在类中。

关键字class后面紧跟类名。Java中定义类名的规则很宽松。名字必须以字母开头，后面可以跟字母和数字的任意组合。长度基本上没有限制。但是不能使用Java保留字（例如，public或class）作为类名（保留字列表请参看附录A）。

从类名FirstSample可以看出，标准的命名规范为：类名是以大写字母开头的名词。如果名字由多个单词组成，每个单词的第一个字母都应该大写（这种在一个单词中间使用大写字母的方式称为骆驼命名法。以其自身为例，应该写成CamelCase）。

源代码的文件名必须与公有类的名字相同，并用.java作为扩展名。因此，存储这段源代码的文件名必须为FirstSample.java（再次提醒大家注意，大小写是非常重要的，千万不能写成firstsample.java）。

如果已经正确地命名了这个文件，并且源代码中没有任何录入错误，在编译这段源代码之后就会得到一个包含这个类字节码的文件。Java编译器将字节码文件自动地命名为FirstSample.class，并与源文件存储在同一个目录下。最后，使用下面这行命令运行这个程序：

```
java FirstSample
```

（请记住，不要添加.class扩展名。）程序执行之后，控制台上将会显示“We will not use ‘Hello,World’ !”。

当使用

```
java ClassName
```

运行编译程序时，Java虚拟机将从指定类中的main方法开始执行（这里的“方法”就是Java中所说的“函数”），因此为了代码能够执行，在类的源文件中必须包含一个main方法。当然，也可以将用户自定义的方法添加到类中，并且在main方法中调用它们（第4章将讲述如何自定义方法）。

注释：根据Java语言规范，main方法必须声明为public（Java语言规范是描述Java语言的官方文档。可以从网站http://java.sun.com/docs/books/jls上阅读或下载）。

不过，当main方法不是public时，有些版本的Java解释器也可以执行Java应用程序。有个程序员报告了这个bug。如果感兴趣的话，可以在网站http://bugs.sun.com/ bugdatabase/index.jsp上输入bug号码4252539查看一下。这个bug被标明“关闭，不予修复。”Sun公司的工程师解释说：Java虚拟机规范（在http://java.sun.com/ docs/books/vmspec）并没有要求main方法一定是public，并且“修复这个bug有可能带来其他的隐患”。好在，这个问题最终得到了解决。在Java SE 1.4及以后的版本中将强制main方法是public的。

从上面这段话可以发现一个问题的两个方面。一方面让质量保证工程师判断在bug报告中是否存在问题是一件很头痛的事情，这是因为其工作量很大，并且工程师对Java的所有细节也未必了解的很清楚。另一方面，Sun公司把bug报告及其解决方案放到网站上让所有人监督检查，这是一种非常了不起的举动。“bug展示”对程序员来说是一种十分有用的资源，甚至程序员可以对感兴趣的bug进行“投票”。得票多的bug在下一个将要发布的JDK版本中得到解决的可能性就大。

需要注意源代码中的括号{ }。在Java中，像在C/C++中一样，用花括号划分程序的各个部分（通常称为块）。Java中任何方法的代码都用“{”开始，用“}”结束。

花括号的使用风格曾经引发过许多无谓的争论。我们的习惯是把匹配的花括号上下对齐。不过，由于空白符会被Java编译器忽略，所以可以选用自己喜欢的任意风格。在下面讲述各种循环语句时，我们还会详细地介绍花括号的使用。

我们暂且不去理睬关键字static void，而仅把它们当作编译Java应用程序必要的部分就行了。在学习完第4章后，这些内容的作用就会揭晓。现在需要记住：每个Java应用程序都必须有一个main方法，其格式如下所示：

```
public class ClassName
{
   public static void main(String[] args)
   {
      program statements
   }
}
```

C++注释：作为一名C++程序员，一定知道类的概念。Java的类与C++的类很相似，但还是有些差异会使人感到困惑。例如，Java中的所有函数都属于某个类的方法（标准术语将其称为方法，而不是成员函数）。因此，Java中的main方法必须有一个外壳类。读者有可能对C++中的静态成员函数（static member functions）十分熟悉。这些成员函数定义在类的内部，并且不对对象进行操作。Java中的main方法必须是静态的。最后，与C/C++一样，关键字void表示这个方法没有返回值，所不同的是main方法没有给操作系统返回“退出代码”。如果main方法正常退出，那么Java应用程序的退出代码为0，表示成功地运行了程序。如果希望在终止程序时返回其他的代码，那就需要调用System.exit方法。

接下来，研究一下这段代码：

```
{
   System.out.println("We will not use 'Hello, World!'");
}
```

一对花括号表示方法体的开始与结束，在这个方法中只包含一条语句。与大多数程序设计语言一样，可以将Java语句看成是这种语言的句子。在Java中，每个句子必须用分号结束。特别需要说明，回车不是语句的结束标志，因此，如果需要可以将一条语句写在多行上。

在上面这个main方法体中只包含了一条语句，其功能是：将一个文本行输出到控制台上。

在这里，使用了System.out对象并调用了它的println方法。注意，点号（·）用于调用方法。Java使用的通用语法是

object.method(parameters)

这等价于函数调用。

在这个示例中，调用了println方法并传递给它一个字符串参数。这个方法将传递给它的字符串参数显示在控制台上。然后，终止这个输出行，以便每次调用println都会在新的一行上显示输出。需要注意一点，Java与C/C++一样，都采用双引号分隔字符串。本章稍后将会详细地讲解有关字符串的知识。

与其他程序设计语言一样，在Java的方法中，可以没有参数，也可以有一个或多个参数（有的程序员把参数叫做变元）。对于一个方法，即使没有参数也需要书写圆括号。例如，不带参数的println方法只打印一个空行。使用下面的语句：

```
System.out.println();
```

注释：System.out还有一个print方法，它在输出之后不换行。例如，System.out.print（“Hello”）打印“Hello”之后不换行，后面的输出紧跟在字符‘o’之后。

3.2 注释

与大多数程序设计语言一样，Java中的注释也不会出现在可执行程序中。因此，可以在源程序中根据需要添加任意多的注释，而不必担心可执行代码会膨胀。在Java中，有三种书写注释的方式。最常用的方式是使用 //，其注释内容从 // 开始到本行结尾。

```
System.out.println("We will not use 'Hello, World!'"); // is this too cute?
```

当需要长篇的注释时，既可以在每行的注释前面标记//，也可以使用/*和*/将一段比较长的注释括起来。请参见例3-1。

例3-1 **FirstSample.java**

```
/**
 * This is the first sample program in Core Java Chapter 3
 * @version 1.01 1997-03-22
 * @author Gary Cornell
 */
public class FirstSample
{
   public static void main(String[] args)
   {
      System.out.println("We will not use 'Hello, World!'");
   }
}
```

第三种注释可以用来自动地生成文档。这种注释以/**开始，以*/结束。有关这种注释的详细内容和自动生成文档的具体方法请参见第4章。

警告：在Java中，/* */注释不能嵌套。也就是说，如果代码本身包含了一个*/，就不能用/*和*/将注释括起来。

3.3 数据类型

Java是一种强类型语言。这就意味着必须为每一个变量声明一种类型。在Java中，一共有8种基本类型（primitive type），其中有4种整型、2种浮点类型、1种用于表示Unicode编码的字符单元的字符类型char（请参见论述char类型的章节）和1种用于表示真值的boolean类型。

注释：Java有一个能够表示任意精度的算术包，通常称为“大数值”（big number）。虽然被称为大数值，但它并不是一种新的Java类型，而是一个Java对象。本章稍后将会详细地介绍它的用法。

3.3.1 整型

整型用于表示没有小数部分的数值，它允许是负数。Java提供了4种整型，具体内容如表3-1所示。

表3-1 Java整型

类　型	存储需求	取值范围
int	4字节	−2 147 483 648～2 147 483 647（正好超过20亿）
short	2字节	−32 768～32 767
long	8字节	−9 223 372 036 854 775 808～9 223 372 036 854 775 807
byte	1字节	−128～127

在通常情况下，int类型最常用。但如果表示星球上的居住人数，就需要使用long类型了。byte和short类型主要用于特定的应用场合，例如，底层的文件处理或者需要控制占用存储空间量的大数组。

在Java中，整型的范围与运行Java代码的机器无关。这就解决了软件从一个平台移植到另一个平台，或者在同一个平台中的不同操作系统之间进行移植给程序员带来的诸多问题。与此相反，C和C++程序需要针对不同的处理器选择最为有效的整型，这样就有可能造成一个在32位处理器上运行很好的C程序在16位系统上运行却发生整数溢出。由于Java程序必须保证在所有机器上都能够得到相同的运行结果，所以每一种数据类型的取值范围必须固定。

长整型数值有一个后缀L（如4000000000L）。十六进制数值有一个前缀0x（如0xCAFE）。八进制有一个前缀0，例如，010对应八进制中的8。很显然，八进制表示法比较容易混淆，所以建议最好不要使用八进制常数。

C++注释：在C和C++中，int表示的整型与目标机器相关。在8086这样的16位处理器上整型数值占2字节；在Sun SPARC这样的32位处理器上，整型数值占4字节；而在Intel Pentium处理器上，C和C++整型依赖于具体的操作系统，对于DOS和Windows 3.1，整型数值占2字节。当Windows程序使用32位模式时，整型数值占4字节。在Java中，所有的数值类型所占据的字节数量与平台无关。

注意，Java没有任何无符号类型（unsigned type）。

3.3.2 浮点类型

浮点类型用于表示有小数部分的数值。在Java中有两种浮点类型，具体内容如表3-2所示。

表3-2 浮点类型

类　型	存储需求	取值范围
float	4字节	大约±3.402 823 47E+38F（有效位数为6～7位）
double	8字节	大约±1.797 693 134 862 315 70E+308（有效位数为15位）

double表示这种类型的数值精度是float类型的两倍（有人称之为双精度数值）。绝大部分应用程序都采用double类型。在很多情况下，float类型的精度很难满足需求。例如，用7位有

效数字足以精确地表示普通雇员的年薪，但表示公司总裁的年薪可能就不够用了。实际上，只有很少的情况适合使用float类型，例如，需要快速地处理单精度数据，或者需要存储大量数据。

float类型的数值有一个后缀F（例如，3.402F）。没有后缀F的浮点数值（如3.402）默认为double类型。当然，也可以在浮点数值后面添加后缀D（例如，3.402D）。

✔ **注释**：在JDK 5.0中，可以使用十六进制表示浮点数值。例如，0.125可以表示成0x1.0p-3。在十六进制表示法中，使用p表示指数，而不是e。注意，尾数采用十六进制，指数采用十进制。指数的基数是2，而不是10。

所有的浮点数值计算都遵循IEEE 754规范。下面是用于表示溢出和出错情况的三个特殊的浮点数值：

- 正无穷大
- 负无穷大
- NaN（不是一个数字）

例如，一个正整数除以0的结果为正无穷大。计算0/0或者负数的平方根结果为NaN。

✔ **注释**：常量Double.POSITIVE_INFINITY、Double.NEGATIVE_INFINITY和Double.NaN（与相应的Float类型的常量一样）分别表示这三个特殊的值，但在实际应用中很少遇到。特别要说明的是，不能这样检测一个特定值是否等于Double.NaN：

```
if (x == Double.NaN) // is never true
```

所有"非数值"的值都认为是不相同的。然而，可以使用Double.isNaN方法：

```
if (Double.isNaN(x)) // check whether x is "not a number"
```

✗ **警告**：浮点数值不适用于禁止出现舍入误差的金融计算中。例如，命令System.out.println(2.0－1.1）将打印出0.8999999999999999，而不是人们想像的0.9。其主要原因是浮点数值采用二进制系统表示，而在二进制系统中无法精确的表示分数1/10。这就好像十进制无法精确地表示1/3一样。如果需要在数值计算中不含有任何舍入误差，就应该使用BigDecimal类，本章稍后将介绍这个类。

3.3.3 char类型

char类型用于表示单个字符。通常用来表示字符常量。例如：'A'是编码为65所对应的字符常量。与"A"不同，"A"是一个包含字符A的字符串。Unicode编码单元可以表示为十六进制值，其范围从\u0000到\Uffff。例如：\u2122表示注册符号，\u03C0表示希腊字母π。

除了可以采用转义序列符\u表示Unicode代码单元的编码之外，还有一些用于表示特殊字符的转义序列符，请参看表3-3。所有这些转义序列符都可以出现在字符常量或字符串的引号内。例如，'\u2122'或"Hello\n"。转义序列符\u还可以出现在字符常量或字符串的引号之外（而其他所有转义序列不可以）。例如：

```
public static void main(String\u005B\u005D args)
```

这种形式完全符合语法规则，\u005B和\u005D是［和］的编码。

表3-3 特殊字符的转义序列符

转义序列	名 称	Unicode值	转义序列	名 称	Unicode值
\b	退格	\u0008	\"	双引号	\u0022
\t	制表	\u0009	\'	单引号	\u0027
\n	换行	\u000a	\\	反斜杠	\u005c
\r	回车	\u000d			

要想弄清char类型，就必须了解Unicode编码表。Unicode打破了传统字符编码方法的限制。在Unicode出现之前，已经有许多种不同的标准：美国的ASCII、西欧语言中的ISO 8859-1、俄国的KOI-8、中国的GB118030和BIG-5等等。这样就产生了下面两个问题：一个是对于任意给定的代码值，在不同的编码方案下有可能对应不同的字母；二是采用大字符集的语言其编码长度有可能不同。例如，有些常用的字符采用单字节编码，而另一些字符则需要两个或更多个字节。

设计Unicode编码的目的就是要解决这些问题。在20世纪80年代开始启动设计工作时，人们认为两个字节的代码宽度足以能够对世界上各种语言的所有字符进行编码，并有足够的空间留给未来的扩展。在1991年发布了Unicode 1.0，当时仅占用65 536个代码值中不到一半的部分。在设计Java时决定采用16位的Unicode字符集，这样会比使用8位字符集的程序设计语言有很大的改进。

十分遗憾，经过一段时间，不可避免的事情发生了。Unicode字符超过了65 536个，其主要原因是增加了大量的汉语、日语和韩国语言中的表意文字。现在，16位的char类型已经不能满足描述所有Unicode字符的需要了。

下面利用一些专用术语解释一下Java语言解决这个问题的基本方法。从JDK 5.0开始。*代码点*（code point）是指与一个编码表中的某个字符对应的代码值。在Unicode标准中，代码点采用十六进制书写，并加上前缀U+，例如U+0041就是字母A的代码点。Unicode的代码点可以分成17个*代码级别*（code plane）。第一个代码级别称为*基本的多语言级别*（basic multilingual plane），代码点从U+0000到U+FFFF，其中包括了经典的Unicode代码；其余的16个附加级别，代码点从U+10000到U+10FFFF，其中包括了一些*辅助字符*（supplementary character）。

UTF-16编码采用不同长度的编码表示所有Unicode代码点。在基本的多语言级别中，每个字符用16位表示，通常被称为*代码单元*（code unit）；而辅助字符采用一对连续的代码单元进行编码。这样构成的编码值一定落入基本的多语言级别中空闲的2048字节内，通常被称为*替代区域*（surrogate area）[U+D800~U+DBFF用于第一个代码单元，U+DC00~U+DFFF用于第二个代码单元]。这样设计十分巧妙，我们可以从中迅速地知道一个代码单元是一个字符的编码，还是一个辅助字符的第一或第二部分。例如，对于整数集合的数学符号，它的代码点是U+1D56B，并且是用两个代码单元U+D835和U+DD6B编码的（有关编码算法的描述请参看http://en.wikipe-dia.org/wiki/UTF-16）。

在Java中，char类型用UTF-16编码描述一个代码单元。

我们强烈建议不要在程序中使用char类型，除非确实需要对UTF-16代码单元进行操作。最好将需要处理的字符串用抽象数据类型表示（有关这方面的内容将在稍后讨论）。

3.3.4 boolean类型

boolean（布尔）类型有两个值：false和true，用来判定逻辑条件。整型值和布尔值之间不能进行相互转换。


C++注释：在C++中，数值或指针可以代替boolean值。整数0相当于布尔值false，非0值相当于布尔值true。在Java中则不行。因此，Java应用程序员不会遇到下述麻烦：

```
if (x = 0) // oops...meant x == 0
```

在C++中这个测试可以编译运行，其结果总是false。而在Java中，这个测试将不能通过编译，其原因是整数表达式x = 0不能转换为布尔值。

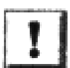
3.4 变量

在Java中，每一个变量属于一种类型（type）。在声明变量时，变量所属的类型位于变量名之前。这里列举一些声明变量的示例：

```
double salary;
int vacationDays;
long earthPopulation;
boolean done;
```

可以看到，每个声明以分号结束。由于声明是一条完整的语句，所以必须以分号结束。

变量名必须是一个以字母开头的由字母或数字构成的序列。需要注意，与大多数程序设计语言相比，Java中“字母”和“数字”的范围要大。字母包括'A'～'Z'、'a'～'z'、'_'或在某种语言中代表字母的任何Unicode字符。例如，德国的用户可以在变量名中使用字母‘ä’；希腊人可以使用π。同样，数字包括'0'～'9'和在某种语言中代表数字的任何Unicode字符。但'+'和'©'这样的符号不能出现在变量名中，空格也不行。变量名中所有的字符都是有意义的，并且大小写敏感。变量名的长度没有限制。


提示：如果想要知道哪些Unicode字符属于Java中的“字母”，可以使用Character类的isJavaIdentifierStart和isJavaIdentifierPart方法进行检测。

另外，不能将变量名命名为Java保留字（请参看附录A中的保留字列表）。

可以在一行中声明多个变量：

```
int i, j; // both are integers
```

不过，不提倡使用这种风格。逐一声明每一个变量可以提高程序的可读性。


注释：如前所述，变量名对大小写敏感，例如，hireday和hireDay是两个不同的变量名。在对两个不同的变量进行命名时，最好不要只存在大小写上的差异。不过，在有些时候，确实很难给变量取一个好的名字。于是，许多程序员将变量名命名为类型名，例如：

```
Box box; // ok--Box is the type and box is the variable name
```

还有一些程序员更加喜欢在变量名前加上前缀“a”：

```
Box aBox;
```

3.4.1 变量初始化

声明一个变量之后，必须用赋值语句对变量进行显式初始化，千万不要使用未被初始化的变量。例如，Java编译器认为下面语句序列是错误的：

```
int vacationDays;
System.out.println(vacationDays); // ERROR--variable not initialized
```

要想对一个已经声明过的变量进行赋值，就需要将变量名放在等号（=）左侧，相应取值的Java表达式放在等号的右侧。

```
int vacationDays;
vacationDays = 12;
```

也可以将变量的声明和初始化放在同一行中。例如：

```
int vacationDays = 12;
```

最后，在Java中可以将声明放在代码中的任何地方。例如，下列代码的书写形式在Java中是完全合法的：

```
double salary = 65000.0;
System.out.println(salary);
int vacationDays = 12; // ok to declare a variable here
```

在Java中，变量的声明尽可能地靠近变量第一次使用的地方，这是一种良好的程序编写风格。



C++注释：C和C++区分变量的声明与定义。例如：

```
int i = 10;
```

是定义一个变量，而

```
extern int i;
```

是声明一个变量。在Java中，不区分变量的声明与定义。

3.4.2 常量

在Java中，利用关键字final声明常量。例如：

```
public class Constants
{
   public static void main(String[] args)
   {
      final double CM_PER_INCH = 2.54;
      double paperWidth = 8.5;
      double paperHeight = 11;
      System.out.println("Paper size in centimeters: "
         + paperWidth * CM_PER_INCH + " by " + paperHeight * CM_PER_INCH);
   }
}
```

关键字final表示这个变量只能被赋值一次。一旦被赋值之后，就不能够再更改了。习惯上，常量名使用大写。

在Java中，经常希望某个常量可以在一个类中的多个方法中使用，通常将这些常量称为类常量。可以使用关键字static final设置一个类常量。下面是使用类常量的示例：

```
public class Constants2
```

```
{
   public static void main(String[] args)
   {
      double paperWidth = 8.5;
      double paperHeight = 11;
      System.out.println("Paper size in centimeters: "
         + paperWidth * CM_PER_INCH + " by " + paperHeight * CM_PER_INCH);
   }

   public static final double CM_PER_INCH = 2.54;
}
```

需要注意，类常量的定义位于main方法的外部。因此，在同一个类的其他方法中也可以使用这个常量。而且，如果一个常量被声明为public，那么其他类的方法也可以使用这个常量。在这个示例中，Constants2.CM_PER-INCH就是这样一个常量。

C++注释：const是Java保留的关键字，但目前并没有使用。在Java中，必须使用final定义常量。

3.5 运算符

在Java中，使用算术运算符＋、－、*、/ 表示加、减、乘、除运算。当参与 / 运算的两个操作数都是整数时，表示整数除法；否则，表示浮点除法。整数的求余操作（有时称为取模）用%表示。例如，15/2等于7，15%2等于1，15.0/2等于7.5。

需要注意，整数被0除将会产生一个异常，而浮点数被0除将会得到无穷大或NaN结果。

可以在赋值语句中采用一种简化的格式书写二元算术运算符。

例如，

```
x += 4;
```

等价于

```
x = x + 4;
```

（通常，将运算符放在赋值号的左侧，如*= 或 %=。）

注释：可移植性是Java语言的设计目标之一。无论在哪个虚拟机上运行，同一运算应该得到同样的结果。对于浮点数的算术运算，实现这样的可移植性是相当困难的。double类型使用64位存储一个double数值，而有些处理器使用80位浮点寄存器。这些寄存器增加了中间过程的计算精度。例如，下列运算：

```
double w = x * y / z;
```

很多Intel处理器计算x * y，并且将结果存储在80位的寄存器中，再除以z并将结果截断为64位。这样可以得到一个更加精确的计算结果，并且还能够避免产生指数溢出。但是，这个结果可能与始终在64位机器上计算的结果不一样。因此，Java虚拟机的最初规范规定所有的中间计算都必须进行截断。这种行为遭到了数值计算团体的反对。截断计算不仅可能导致溢出，而且由于截断操作需要消耗时间，所以在计算速度上要比精确计算慢。为此，Java程序设计语言承认了最优性能与理想结果之间存在的冲突，并给予了改进。在默认情况下，虚拟机设计者允许将中间计算结果采用扩展的精度。但是，对于使用

strictfp关键字标记的方法必须使用严格的浮点计算来产生理想的结果。例如，可以把main方法标记为

```
public static strictfp void main(String[] args)
```

于是，在main方法中的所有指令都将使用严格的浮点计算。如果将一个类标记为strictfp，这个类中的所有方法都要使用严格的浮点计算。

实际的计算方式将取决于Intel处理器。在默认情况下，中间结果允许使用扩展的指数，但不允许使用扩展的尾数（Intel芯片在截断尾数时并不损失性能）。因此，这两种方式的区别仅仅在于采用默认的方式不会产生溢出，而采用严格的计算有可能产生溢出。

如果没有仔细阅读这个注释，也没有什么关系。对大多数程序来说，浮点溢出不属于大问题。在本书中，将不使用strictfp关键字。

3.5.1 自增运算符与自减运算符

当然，程序员都知道加1、减1是数值变量最常见的操作。在Java中，借鉴了C和C++的实现方式，也使用了自增、自减运算符：n++将变量n的当前值加1；n−−将n的值减1。例如：

```
int n = 12;
n++;
```

n的值将变为13。因为这些运算符改变了变量的值，所以它的操作数不能是数值。例如，4++就是一条非法的语句。

实际上，这两个运算符有两种形式。上面介绍的是运算符放在操作数后面的“后缀”形式，还有一种“前缀”形式，++n。两种方式都是对变量值加1。但在表达式中，这两种形式就有区别了。前缀方式先进行加1运算；后缀方式则使用变量原来的值。

```
int m = 7;
int n = 7;
int a = 2 * ++m; // now a is 16, m is 8
int b = 2 * n++; // now b is 14, n is 8
```

我们建议不要在其他表达式的内部使用++，这样编写的代码很容易令人带来迷惑的感觉，并会产生烦人的bug。

当然，++运算符作为C++语言名称的一部分，也引发了有关程序设计语言的第一个笑话。C++的反对者认为这种语言的名称也存在着bug，他们说：“因为只有对它改进之后，我们才有可能使用它，所以它的名字应该命名为++C。”

3.5.2 关系运算符与boolean运算符

Java包含各种关系运算符。其中，使用两个等号 = =检测是否相等。例如，3 = = 7的值为false。

使用 != 检测是否不相等。例如，3 != 7的值为true。

另外，经常使用的运算符还有 < （小于）、> （大于）、<= （小于等于）和>= （大于等于）。

Java沿用了C++的习惯，用&&表示逻辑“与”、用 || 表示逻辑“或”。从!= 运算符很容易看出，!表示逻辑“非”。&&和 || 是按照“短路”方式求值的。如果第一个操作数已经能够确定表达式的值，第二个操作数就不必计算了。如果用 && 对两个表达式进行计算：

expression$_1$ && *expression*$_2$

并且第一个表达式值为false，结果不可能为真。因此，第二个表达式的值就没有必要计算了。这种方式可以避免一些错误的发生。例如，表达式：

```
x != 0 && 1 / x > x + y // no division by 0
```

当x为0时，不会计算第二部分。因此，若x为0，1/x不被计算，也不会出现除以0的错误。

与之类似，对于expression$_1$ || expression$_2$，当第一个表达式为true时，结果自动为true，不必再计算第二部分。

最后，Java支持三元操作?:。在很多时候，这个操作非常有用。表达式

condition ? *expression*$_1$: *expression*$_2$

当条件condition为真时计算第1个表达式，否则计算第2个表达式。例如：

```
x < y ? x : y
```

返回x和y中较小的那个值。

3.5.3 位运算符

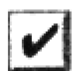
在处理整型数值时，可以直接对组成整型数值的各个位进行操作。这意味着可以使用屏蔽技术获得整数中的各个位。位运算符包括：

& ("与")、| ("或")、^ ("异或")、~ ("非")

这些运算符在位模式下工作。例如，如果n是一个整型变量，并且用二进制表示的n从右数第4位为1，那么

```
int fourthBitFromRight = (n & 8) / 8;
```

返回1；否则返回0。通过运用2的幂次方的&运算可以将其他位屏蔽掉，而只保留其中的某一位。



注释：&和|运算符应用于布尔值，得到的结果也是布尔值。这两个运算符与&&和||的运算非常类似，只是不按“短路”方式计算。即在得到计算结果之前，一定要计算两个操作数的值。

另外，“>>”和“<<”运算符将二进制位进行右移或左移操作。当需要建立位模式屏蔽某些位时，使用这两个运算符十分方便：

```
int fourthBitFromRight = (n & (1 << 3)) >> 3;
```

最后，>>>运算符将用0填充高位；>>运算符用符号位填充高位。没有<<<运算符。



警告：对移位运算符右侧的参数需要进行模32的运算（除非左边的操作数是long类型，在这种情况下需对右侧操作数模64）。例如，1 << 35与1 << 3或8是相同的。



C++注释：在C或C++中无法确定 >> 操作执行的是算术移位（扩展符号位），还是逻辑移位（高位填0）。在执行中将会选择效率较高的一种。这就是说，在C/C++中，>> 运算符实际上只是为非负数定义的。Java消除了这种含糊性。

3.5.4 数学函数与常量

在Math类中，包含了各种各样的数学函数。在编写不同类别的程序时，可能需要的函数也不同。

要想计算一个数值的平方根，可以使用sqrt方法：

```
double x = 4;
double y = Math.sqrt(x);
System.out.println(y); // prints 2.0
```

注释：println方法和sqrt方法存在微小的差异。println方法操作一个定义在System类中的System.out对象。但是，Math类中的sqrt方法操作的不是对象，这样的方法被称为静态方法。有关静态方法的详细内容请参看第4章。

在Java中，没有幂运算，因此需要借助于Math类的pow方法。语句：

```
double y = Math.pow(x, a);
```

将y的值设置为x的a次幂（x^a）。pow方法有两个double类型的参数，其返回结果也为double类型。

Math类提供了一些常用的三角函数：

```
Math.sin
Math.cos
Math.tan
Math.atan
Math.atan2
```

还有指数函数以及它的反函数——自然对数：

```
Math.exp
Math.log
```

最后，Java还提供了两个用于表示π和e常量的近似值：

```
Math.PI
Math.E
```

提示：从JDK 5.0开始，不必在数学方法名和常量名前添加前缀“Math.”，而只要在源文件的顶部加上下列内容就可以了。

例如：

```
import static java.lang.Math.*;
```

在第4章中将讨论静态导入。

```
System.out.println("The square root of \u03C0 is " + sqrt(PI));
```

注释：在Math类中，为了达到最快的性能，所有的方法都使用计算机浮点单元中的例程。如果得到一个完全可预测的结果比运行速度更重要的话，那么就应该使用StrictMath类。它使用“自由发布的Math库”（fdlibm）实现算法，以确保在所有平台上得到相同的结果。有关这些算法的源代码请参看http://www.netlib.org/fdlibm/ index.html（当fdlibm为一个函数提供了多个定义时，StrictMath类就会遵循IEEE 754版本，它的名字将以“e”开头）。

3.5.5 数值类型之间的转换

在程序运行时，经常需要将一种数值类型转换为另一种数值类型。图3-1给出了数值类型之间的合法转换。

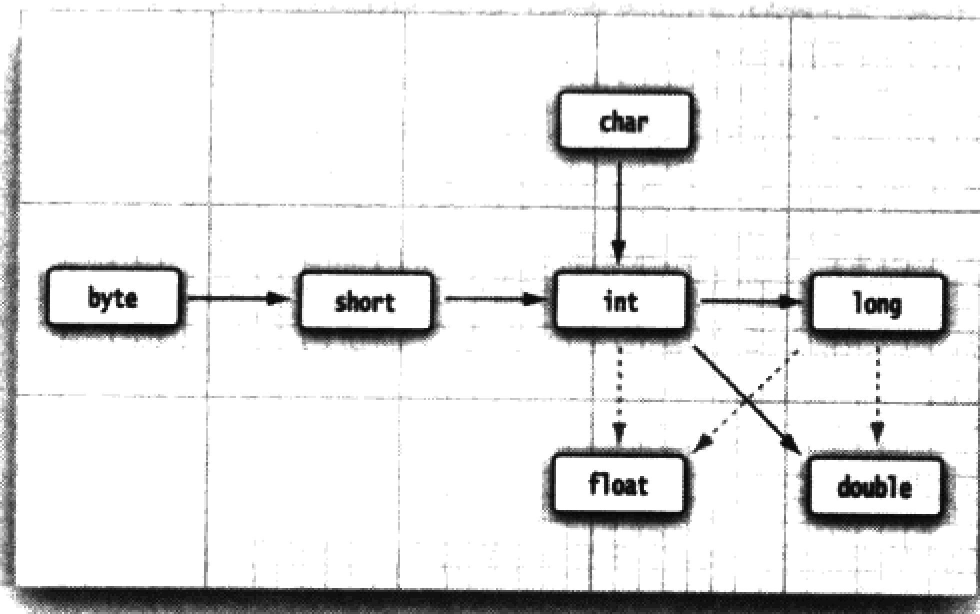


图3-1 数值类型之间的合法转换

在图3-1中有6个实心箭头，表示无信息丢失的转换；有3个虚箭头，表示可能有精度损失的转换。例如，123 456 789是一个大整数，它所包含的位数比float类型所能够表达的位数多。当将这个整型数值转换为float类型时，将会得到同样大小的结果，但却失去了一定的精度。

```
int n = 123456789;
float f = n; // f is 1.23456792E8
```

当使用上面两个数值进行二元操作时（例如n + f，n是整数，f是浮点数），先要将两个操作数转换为同一种类型，然后再进行计算。

- 如果两个操作数中有一个是double类型的，另一个操作数就会转换为double类型。
- 否则，如果其中一个操作数是float类型，另一个操作数将会转换为float类型。
- 否则，如果其中一个操作数是long类型，另一个操作数将会转换为long类型。
- 否则，两个操作数都将被转换为int类型。

3.5.6 强制类型转换

在上一小节中看到，在必要的时候，int类型的值将会自动地转换为double类型。但另一方面，有时也需要将double转换成int。在Java中，允许进行这种数值之间的类型转换。当然，有可能会丢失一些信息。在这种情况下，需要通过*强制类型转换*（cast）实现这个操作。强制类型转换的语法格式是在圆括号中给出想要转换的目标类型，后面紧跟待转换的变量名。例如：

```
double x = 9.997;
int nx = (int) x;
```

这样，变量nx的值为9。强制类型转换通过截断小数部分将浮点值转换为整型。

如果想对浮点数进行舍人运算，以便得到*最接近的*整数（在很多情况下，希望使用这种操

作方式），那就需要使用Math.round方法：

```
double x = 9.997;
int nx = (int) Math.round(x);
```

现在，变量nx的值为10。当调用round的时候，仍然需要使用强制类型转换（int）。其原因是round方法返回的结果为long类型，由于存在信息丢失的可能性，所以只有使用显式的强制类型转换才能够将long类型转换成int类型。

警告：如果试图将一个数值从一种类型强制转换为另一种类型，而又超出了目标类型的表示范围，结果就会截断成一个完全不同的值。例如，(byte) 300的实际值为44。

C++注释：不要在boolean类型与任何数值类型之间进行强制类型转换，这样可以防止发生错误。只有极少数的情况才需要将布尔类型转换为数值类型，这时可以使用条件表达式b? 1:0。

3.5.7 括号与运算符级别

表3-4给出了运算符的优先级。如果不使用圆括号，就按照给出的运算符优先级次序进行计算。同一个级别的运算符按照从左到右的次序进行计算（除了表中给出的右结合运算符外。）例如，由于 && 的优先级比 || 的优先级高，所以表达式

```
a && b || c
```

等价于

```
(a && b) || c
```

又因为 += 是右结合运算符，所以表达式

```
a += b += c
```

等价于

```
a += (b += c)
```

也就是将b += c 的结果（加上c之后的b）加到a上。

C++注释：与C或C++不同，Java不使用逗号运算符。不过，可以在for语句中使用逗号分隔表达式列表。

表3-4 运算符优先级

运算符	结合性
[ ] . ( )（方法调用）	从左向右
! ~ ++ -- +（一元运算）-（一元运算）( )（强制类型转换）new	从右向左
* / %	从左向右
+ -	从左向右
<< >> >>>	从左向右
< <= > >= instanceof	从左向右
== !=	从左向右
&	从左向右
^	从左向右

（续）

运算符	结合性
|	从左向右
&&	从左向右
||	从左向右
?:	从右向左
= += -= *= /= %= &= |= ^= <<= >>= >>>=	从右向左

3.5.8 枚举类型

有时候，变量的取值只在一个有限的集合内。例如：销售的服装或比萨饼只有小、中、大和超大这四种尺寸。当然，可以将这些尺寸分别编码为1、2、3、4或S、M、L、X。但这样存在着一定的隐患。在变量中很可能保存的是一个错误的值（如0或m）。

从JDK 5.0开始，针对这种情况，可以自定义枚举类型。枚举类型包括有限个命名的值。例如，

```
enum Size { SMALL, MEDIUM, LARGE, EXTRA_LARGE };
```

现在，可以声明这样一种类型的变量：

```
Size s = Size.MEDIUM;
```

Size类型的变量只能存储这个类型声明中给定的某个枚举值，或者null值，null表示这个变量没有设置任何值。

有关枚举类型的详细内容将在第5章介绍。

3.6 字符串

从概念上讲，Java字符串就是Unicode字符序列。例如，串“Java\u2122”由5个Unicode字符J、a、v、a和™。Java没有内置的字符串类型，而是在标准Java类库中提供了一个预定义类String。每个用双引号括起来的字符串都是String类的一个实例：

```
String e = ""; // an empty string
String greeting = "Hello";
```

3.6.1 子串

String类的substring方法可以从一个较大的字符串提取出一个子串。例如：

```
String greeting = "Hello";
String s = greeting.substring(0, 3);
```

创建了一个由字符“Hel”组成的字符串。

substring方法的第二个参数是不想复制的第一个位置。这里要复制位置为0、1和2（从0到2，包括0和2）的字符。在substring中从0开始计数，直到3为止，但不包含3。

substring的工作方式有一个优点：容易计算子串的长度。字符串s.substring(a, b)的长度为b-a。例如，子串“Hel”的长度为3－0＝3。

3.6.2 拼接

与绝大多数的程序设计语言一样，Java语言允许使用+号连接（拼接）两个字符串。

```
String expletive = "Expletive";
String PG13 = "deleted";
String message = expletive + PG13;
```

上述代码将“Expletivedeleted”赋给变量message（注意，单词之间没有空格，+号按照给定的次序将两个字符串拼接起来）。

当将一个字符串与一个非字符串的值进行拼接时，后者被转换成字符串（在第5章中可以看到，任何一个Java对象都可以转换成字符串）。例如：

```
int age = 13;
String rating = "PG" + age;
```

rating得到“PG13”。

这种特性通常用在输出语句中。例如：

```
System.out.println("The answer is " + answer);
```

这是一条合法的语句，并且将会打印出所希望的结果（单词is后面加了一个空格，输出时也会加上这个空格）。

3.6.3 不可变字符串

String类没有提供用于修改字符串的方法。如果希望将greeting的内容修改为“Help!”，不能直接地将greeting的最后两个位置的字符修改为‘p’和‘!’。这对于C 程序员来说，将会感到无从下手。如何修改这个字符串呢？在Java中实现这项操作非常容易。首先提取需要的字符，然后再拼接上替换的字符串：

```
greeting = greeting.substring(0, 3) + "p!";
```

上面这条语句将greeting当前值修改为“Help! ”。

由于不能修改Java字符串中的字符，所以在Java文档中将String类对象称为不可变字符串，如同数字3永远是数字3一样，字符串“Hello”永远包含字符H、e、l、l和o的代码单元序列，而不能修改其中的任何一个字符。当然，可以修改字符串变量greeting，让它引用另外一个字符串，这就如同可以将存放3的数值变量改成存放4一样。

这样做是否会降低运行效率呢？看起来好像修改一个代码单元要比创建一个新字符串更加简洁。答案是：也对，也不对。的确，通过拼接“Hel”和“p!”来创建一个新字符串的效率确实不高。但是，不可变字符串却有一个优点：编译器可以让字符串共享。

为了弄清具体的工作方式，可以想像将各种字符串存放在公共的存储池中。字符串变量指向存储池中相应的位置。如果复制一个字符串变量，原始字符串与复制的字符串共享相同的字符。

总而言之，Java的设计者认为共享带来的高效率远远胜过于提取、拼接字符串所带来的低效率。查看一下程序会发现：很少需要修改字符串，而是往往需要对字符串进行比较（有一种例外情况，将源自于文件或键盘的单个字符或较短的字符串汇集成字符串。为此，Java提供了

一个独立的类，在“构建字符串”中将详细地给予介绍）。

C++注释：在C程序员第一次接触Java字符串的时候，常常会感到迷惑，因为他们总将字符串认为是字符型数组：

```
char greeting[] = "Hello";
```

这种认识是错误的，Java字符串更加像char*指针，

```
char* greeting = "Hello";
```

当采用另一个字符串替换greeting的时候，Java代码主要进行下列操作：

```
char* temp = malloc(6);
strncpy(temp, greeting, 3);
strncpy(temp + 3, "p!", 3);
greeting = temp;
```

的确，现在greeting指向字符串“Help!”。即使一名最顽固的C程序员也得承认Java语法要比一连串的strncpy调用舒适得多。然而，如果将greeting赋予另外一个值又会怎样呢？

```
greeting = "Howdy";
```

这样做会不会产生内存遗漏呢？毕竟，原始字符串放置在堆中。十分幸运，Java将自动地进行垃圾回收。如果一块内存不再使用了，系统最终会将其回收。

对于一名使用ANSI C++定义的string类的C++程序员，会感觉使用Java的String类型更为舒适。C++ string对象也自动地进行内存的分配与回收。内存管理是通过构造器、赋值操作和析构器显式执行的。然而，C++字符串是可修改的，也就是说，可以修改字符串中的单个字符。

3.6.4 检测字符串是否相等

可以使用equals方法检测两个字符串是否相等。对于表达式：

```
s.equals(t)
```

如果字符串s与字符串t相等，则返回true；否则，返回false。需要注意，s与t可以是字符串变量，也可以是字符串常量。例如，下列表达式是合法的：

```
"Hello".equals(greeting)
```

要想检测两个字符串是否相等，而不区分大小写，可以使用equalsIgnoreCase方法。

```
"Hello".equalsIgnoreCase("hello")
```

一定不能使用 = = 运算符检测两个字符串是否相等！这个运算符只能够确定两个字符串是否放置在同一个位置上。当然，如果字符串放置在同一个位置上，它们必然相等。但是，完全有可能将内容相同的多个字符串的拷贝放置在不同的位置上。

```
String greeting = "Hello"; //initialize greeting to a string
if (greeting == "Hello") . . .
   // probably true
if (greeting.substring(0, 3) == "Hel") . . .
   // probably false
```

如果虚拟机始终将相同的字符串共享，就可以使用 == 运算符检测是否相等。但实际上只有字符串常量是共享的，而+或substring等操作产生的结果并不是共享的。因此，千万不要使

用 == 运算符测试字符串的相等性，以免在程序中出现糟糕的bug。从表面上看，这种bug很像随机产生的间歇性错误。

C++注释：对于习惯使用C++的string类的人来说，在进行相等性检测的时候一定要特别小心。C++的string类重载了==运算符以便检测字符串内容的相等性。可惜Java没有采用这种方式，它的字符串"看起来、感觉起来"与数值一样，但进行相等性测试时，其操作方式又类似于指针。语言的设计者本应该像对+那样也进行特殊处理，即重定义 == 运算符。当然，每一种语言都会存在一些不太一致的地方。

C程序员从不使用 == 对字符串进行比较，而使用strcmp函数。Java的compareTo方法与strcmp完全类似，因此，可以这样使用：

```
if (greeting.compareTo("Hello") == 0) . . .
```

不过，使用equals看起来更为清晰。

3.6.5 代码点与代码单元

Java字符串由char序列组成。从前面已经看到，字符数据类型是一个采用UTF-16编码表示Unicode代码点的代码单元。大多数的常用Unicode字符使用一个代码单元就可以表示，而辅助字符需要一对代码单元表示。

length方法将返回采用UTF-16编码表示的给定字符串所需要的代码单元数量。例如：

```
String greeting = "Hello";
int n = greeting.length(); // is 5.
```

要想得到实际的长度，即代码点数量，可以调用：

```
int cpCount = greeting.codePointCount(0, greeting.length());
```

调用s.charAt(n) 将返回位置n的代码单元，n介于0～s.length()-1之间。例如：

```
char first = greeting.charAt(0); // first is 'H'
char last = greeting.charAt(4); // last is 'o'
```

要想得到第i个代码点，应该使用下列语句

```
int index = greeting.offsetByCodePoints(0, i);
int cp = greeting.codePointAt(index);
```

注释：Java以独特的风格对字符串中的代码单元计数：字符串中的第一个代码单元位置为0。这种习惯起源于C，这样处理主要出于技术上的原因。具体理由似乎已经淡忘，而麻烦却保留了下来。但是，许多程序员习惯于这种风格，因而Java设计者也就将其保留了下来。

为什么会对代码单元如此大惊小怪？请考虑下列语句：

```
ℤ is the set of integers
```

使用UTF-16编码表示 ℤ 需要两个代码单元。调用

```
char ch = sentence.charAt(1)
```

返回的不是空格，而是第二个代码单元 ℤ 。为了避免这种情况的发生，请不要使用char类型。这太低级了。

如果想要遍历一个字符串，并且依次查看每一个代码点，可以使用下列语句：

```
int cp = sentence.codePointAt(i);
if (Character.isSupplementaryCodePoint(cp)) i += 2;
else i++;
```

非常幸运，codePointAt方法能够辨别一个代码单元是辅助字符的第一部分还是第二部分，并能够返回正确的结果。也就是说，可以使用下列语句实现回退操作：

```
i--;
int cp = sentence.codePointAt(i);
if (Character.isSupplementaryCodePoint(cp)) i--;
```

3.6.6 字符串API

Java中的String类包含了50多个方法。令人惊讶的是绝大多数都很有用，可以设想使用的频繁非常高。下面的API注释汇总了一部分最常用的方法。

注释：可以发现，本书中给出的API注释会有助于理解Java应用程序编程接口（API）。每一个API的注释都以形如java.lang.String的类名开始。java.lang包的重要性将在第4章给予解释。类名之后是一个或多个方法的名字、解释和参数描述。

在这里，一般不列出某个类的所有方法，而是选择一些使用最频繁的方法，并以简洁的方式给予描述。完整的方法列表请参看联机文档（请参看Reading the On-Line API Documentation）。这里还列出了所给类的版本号。如果某个方法是在这个版本之后添加的，就会给出一个单独的版本号。

API java.lang.string 1.0

- `char charAt (int index)`
  返回给定位置的代码单元。除非对底层的代码单元感兴趣，否则不需要调用这个方法。
- `int codePointAt(int index)` 5.0
  返回从给定位置开始或结束的代码点。
- `int offsetByCodePoints(int startIndex, int cpCount)` 5.0
  返回从startIndex代码点开始，位移cpCount后的代码点索引。
- `int compareTo(String other)`
  按照字典顺序，如果字符串位于other之前，返回一个负数；如果字符串位于other之后，返回一个正数；如果两个字符串相等，返回0。
- `boolean endsWith(String suffix)`
  如果字符串以suffix结尾，返回true。
- `boolean equals(Object other)`
  如果字符串与other相等，返回true。
- `boolean equalsIgnoreCase(String other)`
  如果字符串与other相等（忽略大小写），返回true。
- `int indexOf(String str)`
- `int indexOf(String str, int fromIndex)`

- int indexOf(int cp)
- int indexOf(int cp, int fromIndex)

  返回与字符串str或代码点cp匹配的的第一个子串的开始位置。这个位置从索引0或fromIndex开始计算。如果在原始串中不存在str，返回－1。
- int lastIndexOf(String str)
- int lastIndexOf(String str, int fromIndex)
- int lastindexOf(int cp)
- int lastindexOf(int cp, int fromIndex)

  返回与字符串str或代码点cp匹配的最后一个子串的开始位置。这个位置从原始串尾端或fromIndex开始计算。
- int length( )

  返回字符串的长度。
- int codePointCount(int startIndex, int endIndex) 5.0

  返回startIndex和endIndex－1之间的代码点数量。没有配成对的代用字符将计入代码点。
- String replace(CharSequence oldString,CharSequence newString)

  返回一个新字符串。这个字符串用newString代替原始字符串中所有的oldString。可以用String或StringBuilder对象作为CharSequence参数。
- boolean startsWith(String prefix)

  如果字符串以preffix字符串开始，返回true。
- String substring(int beginIndex)
- String substring(int beginIndex, int endIndex)

  返回一个新字符串。这个字符串包含原始字符串中从beginIndex到串尾或endIndex-1的所有代码单元。
- String toLowerCase( )

  返回一个新字符串。这个字符串将原始字符串中的所有大写字母改成了小写字母。
- String toUpperCase( )

  返回一个新字符串。这个字符串将原始字符串中的所有小写字母改成了大写字母。
- String trim( )

  返回一个新字符串。这个字符串将删除了原始字符串头部和尾部的空格。

3.6.7 阅读联机API文档

正如前面所看到的，String类包含许多方法。而且，在标准库中有几千个类，方法数量更加惊人。要想记住所有的类和方法是一件不太不可能的事情。因此，学会使用在线API文档十分重要，从中可以查阅到标准类库中的所有类和方法。API文档是JDK的一部分，它是HTML格式的。让浏览器指向安装JDK的docs/api/index.html子目录，就可以看到如图3-2所示的屏幕。

可以看到，屏幕被分成三个窗框。在左上方的小窗框中显示了可使用的所有包。在它下面稍大的窗框中列出了所有的类。点击任何一个类名之后，这个类的API文档就会显示在右侧的

大窗框中（请参看图3-3）。例如，要获得有关String类方法的更多信息，可以滚动第二个窗框，直到看见String链接为止，然后点击这个链接。

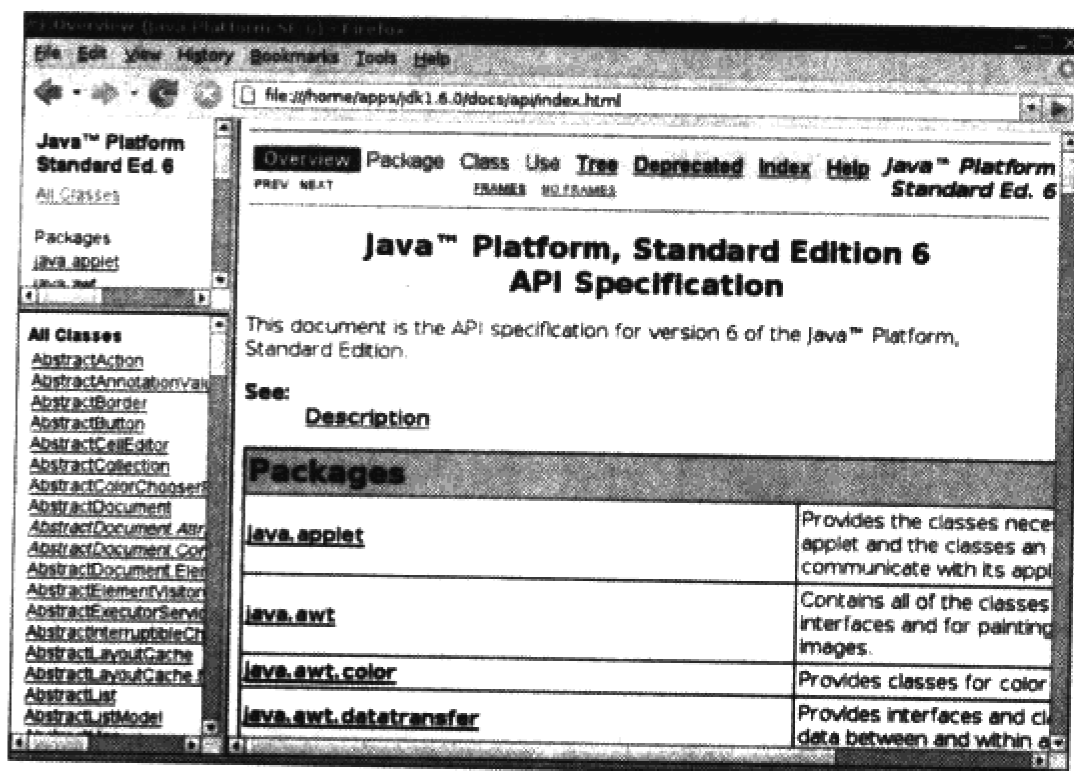


图3-2 API文档的三个窗格

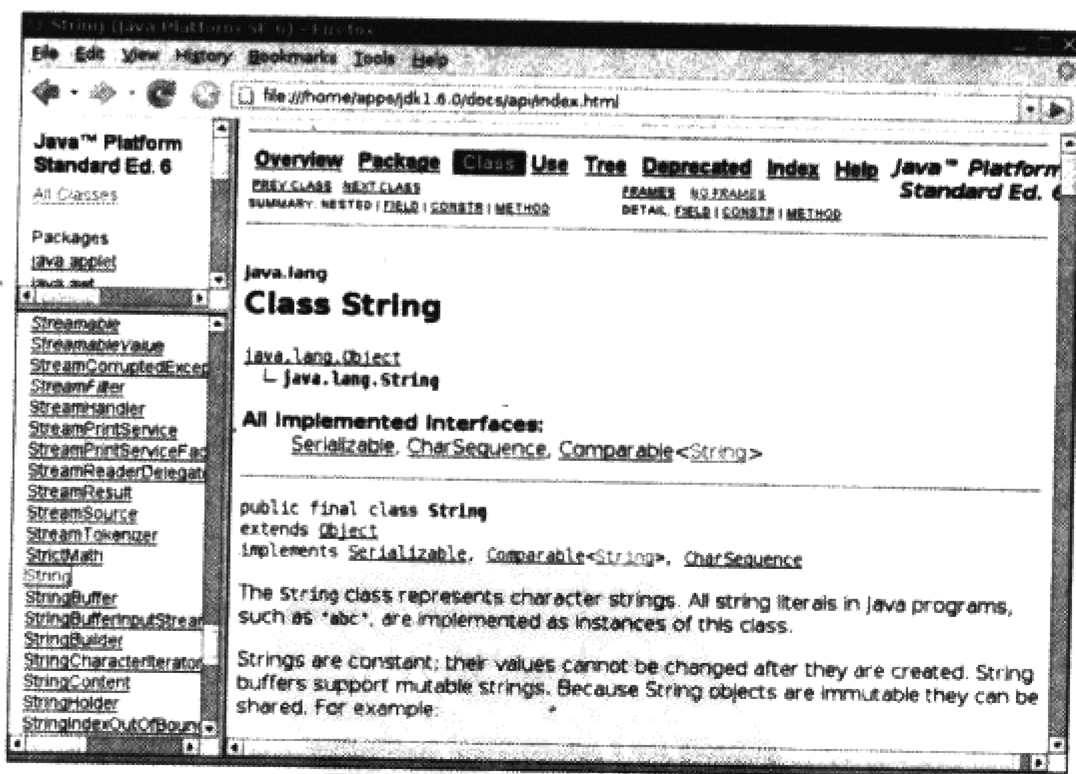


图3-3 String类的描述

接下来，滚动右面的窗框，直到看见按字母顺序排列的所有方法为止（请参看图3-4）。点击任何一个方法名便可以查看这个方法的详细描述（参见图3-5）。例如，如果点击compareToIgnoreCase链接，就会看到compareToIgnoreCase方法的描述。

! **提示**：马上在浏览器中将docs/api/index.html页面做一个书签。

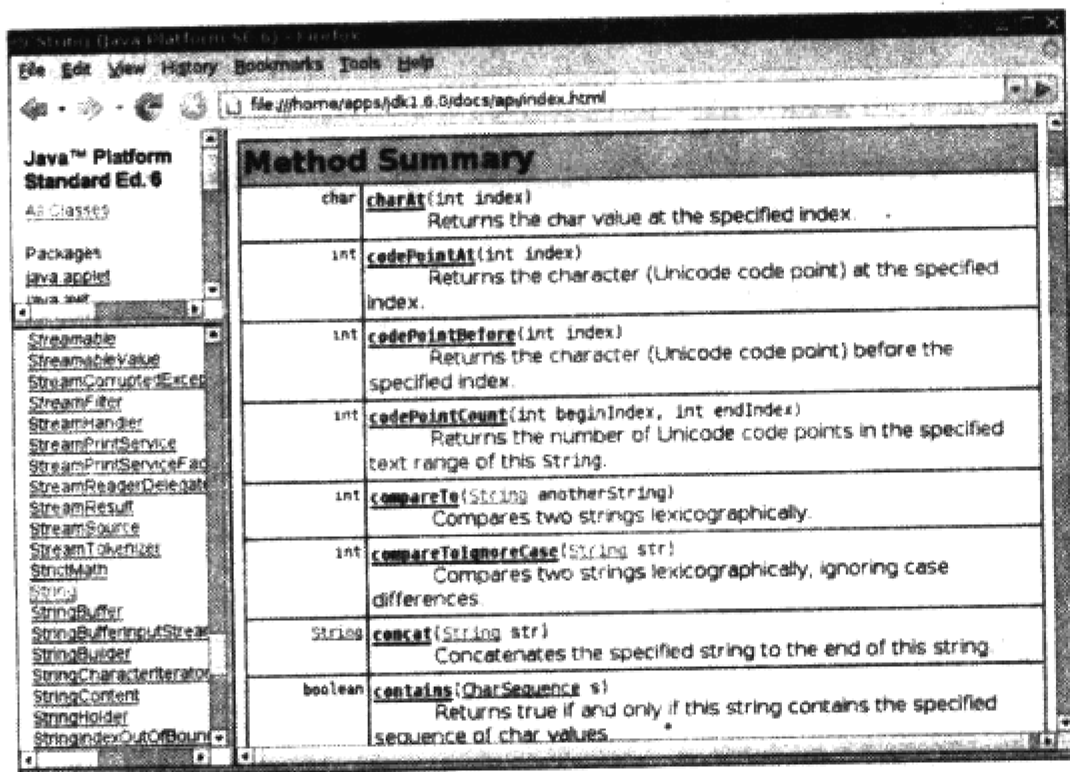


图3-4 String类方法的小结

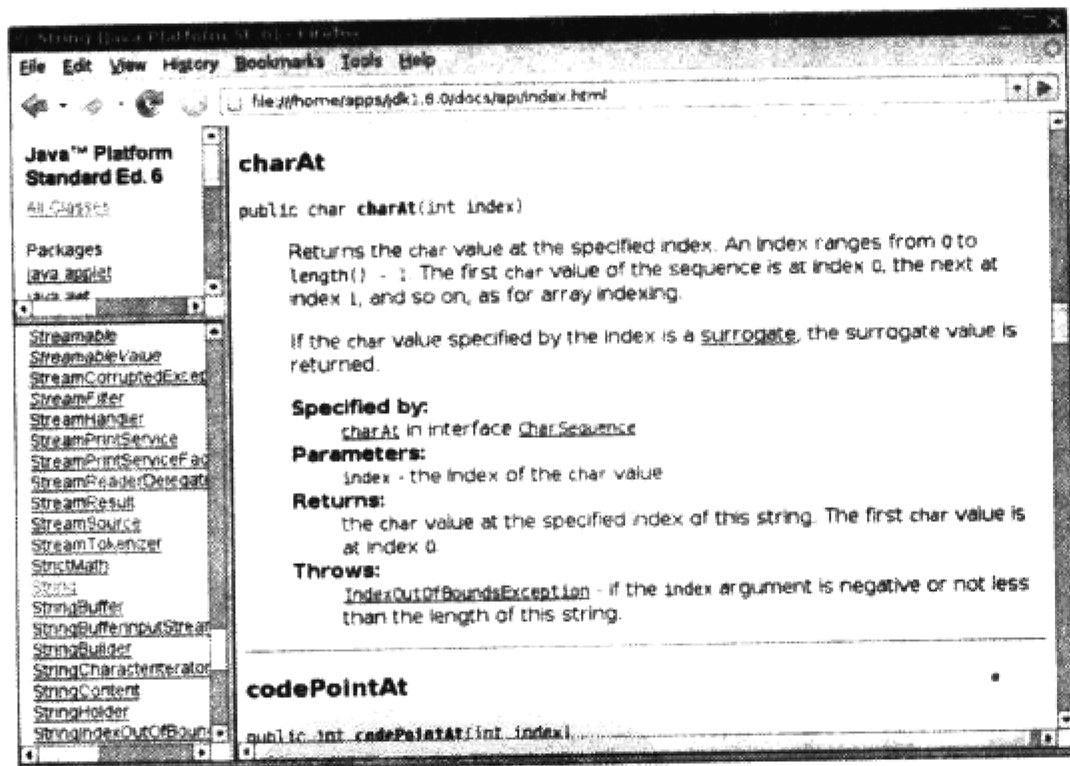


图3-5 String方法的详细描述

3.6.8 构建字符串

有些时候，需要由较短的字符串构建字符串，例如，按键或来自文件中的单词。采用字符串连接的方式达到此目的效率比较低。每次连接字符串，都会构建一个新的String对象，既耗时，又浪费空间。使用StringBuilder类就可以避免这个问题的发生。

如果需要用许多小段的字符串构建一个字符串，那么应该按照下列步骤进行。首先，构建一个空的字符串构建器：

```
StringBuilder builder = new StringBuilder();
```

(有关构造器和new操作符的内容将在第4章详细地介绍)。

当每次需要添加一部分内容时，就调用append方法。

```
builder.append(ch); // appends a single character
builder.append(str); // appends a string
```

在需要构建字符串时就调用toString方法，将可以得到一个String对象，其中包含了构建器中的字符序列。

```
String completedString = builder.toString();
```

注释：在JDK5.0中引入StringBuilder类。这个类的前身是StringBuffer，其效率略微有些低，但允许采用多线程的方式执行添加或删除字符的操作。如果所有字符串在一个单线程中（通常都是这样）编辑，则应该用StringBuilder替代它。这两个类的API是相同的。

下面的API注解包含了StringBuilder类中的重要方法。

API java.lang.StringBuilder 5.0

- `StringBuilder()`
  构造一个空的字符串构建器。
- `int length()`
  返回构建器或缓冲器中的代码单元数量。
- `StringBuilder append(String str)`
  追加一个字符串并返回this。
- `StringBuilder append(char c)`
  追加一个代码单元并返回this。
- `StringBuilder appendCodePoint(int cp)`
  追加一个代码点，并将其转换为一个或两个代码单元并返回this。
- `void setCharAt(int i,char c)`
  将第i个代码单元设置为c。
- `StringBuilder insert(int offset,String str)`
  在offset位置插入一个字符串并返回this。
- `StringBuilder insert(int offset,Char c)`
  在offset位置插入一个代码单元并返回this。
- `StringBuilder delete(int startIndex,int endIndex)`
  删除偏移量从startIndex到－endIndex－1的代码单元并返回this。
- `String toString()`
  返回一个与构建器或缓冲器内容相同的字符串。

3.7 输入输出

为了增加后面示例程序的趣味性，需要程序能够接收输入，并以适当的格式输出。当然，现代的程序都使用GUI收集用户的输入，然而，编写这种界面的程序需要使用较多的工具与技

术，目前还不具备这些条件。主要原因是需要熟悉Java程序设计语言，因此只要有简单的用于输入输出的控制台就可以了。第7章～第9章将详细地介绍GUI程序设计。

3.7.1 读取输入

前面已经看到，打印输出到“标准输出流”（即控制台窗口）是一件非常容易的事情，只要调用System.out.println即可。然而，读取“标准输入流”System.in就没有那么简单了。要想通过控制台进行输入，首先需要构造一个Scanner对象，并与“标准输入流”System.in关联。

```
Scanner in = new Scanner(System.in);
```

（构造器和new操作符将在第4章中详细地介绍。）

现在，就可以使用Scanner类的各种方法实现输入操作了。例如，nextLine方法将输入一行。

```
System.out.print("What is your name? ");
String name = in.nextLine();
```

在这里，使用nextLine方法是因为在输入行中有可能包含空格。要想读取一个单词（以空白符作为分隔符），就调用

```
String firstName = in.next();
```

要想读取一个整数，就调用nextInt方法。

```
System.out.print("How old are you? ");
int age = in.nextInt();
```

与此类似，要想读取下一个浮点数，就调用nextDouble方法。

在例3-2的程序中，询问用户姓名和年龄，然后打印一条如下格式的消息：

```
Hello, Cay. Next year, you'll be 46
```

最后，在程序的最开始添加上一行：

```
import java.util.*;
```

Scanner类定义在java.util包中。当使用的类不是定义在基本java.lang包中时，一定要使用import指示字将相应的包加载进来。有关包与import指示字的详细描述请参看第4章。

例3-2 InputTest.java

```
import java.util.*;

/**
 * This program demonstrates console input.
 * @version 1.10 2004-02-10
 * @author Cay Horstmann
 */
public class InputTest
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);

      // get first input
      System.out.print("What is your name? ");
      String name = in.nextLine();

```

```
        // get second input
        System.out.print("How old are you? ");
        int age = in.nextInt();

        // display output on console
        System.out.println("Hello, " + name + ". Next year, you'll be " + (age + 1));
    }
}
```

注释： 因为输入是可见的，所以Scanner类不适用于从控制台读取密码。Java SE 6特别引入了Console类实现这个目的。要想读取一个密码，可以采用下列代码：

```
Console cons = System.console();
String username = cons.readLine("User name: ");
char[] passwd = cons.readPassword("Password: ");
```

为了安全起见，返回的密码存放在一维字符数组中，而不是字符串中。在对密码进行处理之后，应该马上用一个填充值覆盖数组元素（数组处理将在本章稍后介绍）。

采用Console对象处理输入不如采用Scanner方便。每次只能读取一行输入，而没有能够读取一个单词或一个数值的方法。

API `java.util.Scanner` 5.0

- `Scanner (InputStream in)`

  用给定的输入流创建一个Scanner对象。
- `String nextLine( )`

  读取输入的下一行内容。
- `String next( )`

  读取输入的下一个单词（以空格作为分隔符）。
- `int nextInt( )`
- `double nextDouble( )`

  读取并转换下一个表示整数或浮点数的字符序列。
- `boolean hasNext( )`

  检测输入中是否还有其他单词。
- `boolean hasNextInt( )`
- `boolean hasNextDouble( )`

  检测是否还有表示整数或浮点数的下一个字符序列。

API `java.lang.System` 1.0

- `static Console console( )` 6

  如果有可能进行交互操作，就通过控制台窗口为交互的用户返回一个Console对象，否则返回null。对于任何一个通过控制台窗口启动的程序，都可使用Console对象。否则，其可用性将与所使用的系统有关。

API java.io.Console 6

- static char[] readPassword(String prompt, Object...args)
- static String readLine(String prompt, Object...args)

  显示字符串prompt并且读取用户输入，直到输入行结束。args参数可以用来提供输入格式。有关这部分内容将在下一节中介绍。

3.7.2 格式化输出

可以使用System.out.print(x) 将数值x输出到控制台上。这条命令将以x对应的数据类型所允许的最大非0数字位数打印输出x。例如：

```
double x = 10000.0 / 3.0;
System.out.print(x);
```

打印

```
3333.3333333333335
```

如果希望显示美元、美分等符号，则有可能会出现问题。

在早期的Java版本中，格式化数值曾引起过一些争议。庆幸的是，Java SE 5.0沿用了C语言库函数中的printf方法。例如，调用

```
System.out.printf("%8.2f", x);
```

可以用8个字符的宽度和小数点后两个字符的精度打印x。也就是说，打印输出一个空格和7个字符，如下所示：

```
3333.33
```

在printf中，可以使用多个参数，例如：

```
System.out.printf("Hello, %s. Next year, you'll be %d", name, age);
```

每一个以%字符开始的格式说明符都用相应的参数替换。格式说明符尾部的转换符将指示被格式化的数值类型：f表示浮点数，s表示字符串，d表示十进制整数。表3-5列出了所有转换符。

表3-5 用于printf的转换符

转换符	类型	举例	转换符	类型	举例
d	十进制整数	159	s	字符串	Hello
x	十六进制整数	9f	c	字符	H
o	八进制整数	237	b	布尔	True
f	定点浮点数	15.9	h	散列码	42628b2
e	指数浮点数	1.59e+01	tx	日期时间	见表3-7
g	通用浮点数	—	%	百分号	%
a	十六进制浮点数	0x1.fccdp3	n	与平台有关的行分隔符	—

另外，还可以给出控制格式化输出的各种标志。表3-6列出了所有的标志。例如，逗号标志增加了分组的分隔符。即

```
System.out.printf("%,.2f", 10000.0 / 3.0);
```

打印

```
3,333.33
```

可以使用多个标志，例如，“%,(.2f” 使用分组的分隔符并将负数括在括号内。

表3-6 用于printf的标志

标 志	目 的	举 例
+	打印正数和负数的符号	+3333.33
空格	在正数之前添加空格	| 3333.33|
0	数字前面补0	003333.33
-	左对齐	|3333.33 |
(	将负数括在括号内	(3333.33)
,	添加分组分隔符	3,333.33
# (对于f格式)	包含小数点	3,333
# (对于x或0格式)	添加前缀0x或0	0xcafe
$	给定被格式化的参数索引。例如，%1$d，%1$x将以十进制和十六进制格式打印第1个参数	159 9F
<	格式化前面说明的数值。例如，%d%<x以十进制和十六进制打印同一个数值	159 9F

注释：可以使用s转换符格式化任意的对象。对于任意实现了Formattable接口的对象都将调用formatTo方法；否则将调用toString方法，它可以将对象转换为字符串。在第5章中将讨论toString方法，在第6章中将讨论接口。

可以使用静态的String.format方法创建一个格式化的字符串，而不打印输出：

```
String message = String.format("Hello, %s. Next year, you'll be %d", name, age);
```

尽管在第4章之前，没有对Date类型进行过详细地描述，但基于完整性的考虑，还是简略地介绍一下printf方法中日期与时间的格式化选项。在这里，使用以t开始，以表3-7中任意字母结束的两个字母格式。例如，

```
System.out.printf("%tc", new Date());
```

这条语句将用下面的格式打印当前的日期和时间：

```
Mon Feb 09 18:05:19 PST 2004
```

表3-7 日期和时间的转换符

转换符	类 型	举 例
c	完整的日期和时间	Mon Feb 09 18:05:19 PST 2004
F	ISO 8601日期	2004-02-09
D	美国格式的日期（月/日/年）	02/09/2004
T	24小时时间	18:05:19
r	12小时时间	06:05:19 pm
R	24小时时间没有秒	18:05
Y	4位数字的年（前面补0）	2004

（续）

转换符	类型	举例
y	年的后两位数字（前面补0）	04
C	年的前两位数字（前面补0）	20
B	月的完整拼写	February
b或h	月的缩写	Feb
m	两位数字的月（前面补0）	02
d	两位数字的日（前面补0）	09
e	两位数字的月（前面不补0）	9
A	星期几的完整拼写	Monday
a	星期几的缩写	Mon
j	三位数的年中的日子（前面补0），在001到366之间	069
H	两位数字的小时（前面补0），在0到23之间	18
k	两位数字的小时（前面不补0），在0到23之间	18
I	两位数字的小时（前面补0），在0到12之间	06
l	两位数字的小时（前面不补0），在0到12之间	6
M	两位数字的分钟（前面补0）	05
S	两位数字的秒（前面补0）	19
L	三位数字的毫秒（前面补0）	047
N	九位数字的毫微秒（前面补0）	047000000
P	上午或下午的大写标志	PM
p	上午或下午的小写标志	pm
z	从GMT起，RFC822数字位移	−0800
Z	时区	PST
s	从格林威治时间1970-01-01 00:00:00起的秒数	1078884319
Q	从格林威治时间1970-01-01 00:00:00起的毫秒数	1078884319047

从表3-7可以看到，某些格式只给出了指定日期的部分信息。例如，只有日期或月份。如果需要多次对日期操作才能实现对每一部分进行格式化的目的就太笨拙了。为此，可以采用一个格式化的字符串指出要被格式化的参数索引。索引必须紧跟在%后面，并以$终止。例如，

```
System.out.printf("%1$s %2$tB %2$te, %2$tY", "Due date:", new Date());
```

打印

```
Due date: February 9, 2004
```

还可以选择使用<标志。它指示前面格式说明中的参数将被再次使用。也就是说，下列语句将产生与前面语句同样的输出结果。

```
System.out.printf("%s %tB %<te, %<tY", "Due date:", new Date());
```

提示：参数索引值从1开始，而不是从0开始，%1$...对第1个参数格式化。这就避免了与0标志混淆。

现在，已经了解了printf方法的所有特性。图3-6给出了格式说明符的语法图。

注释：许多格式化规则是本地环境特有的。例如，在德国，十进制的分隔符是句号而不是逗号，Monday被格式化为Montag。在卷II中将介绍如何控制应用程序与地域有关的行为。

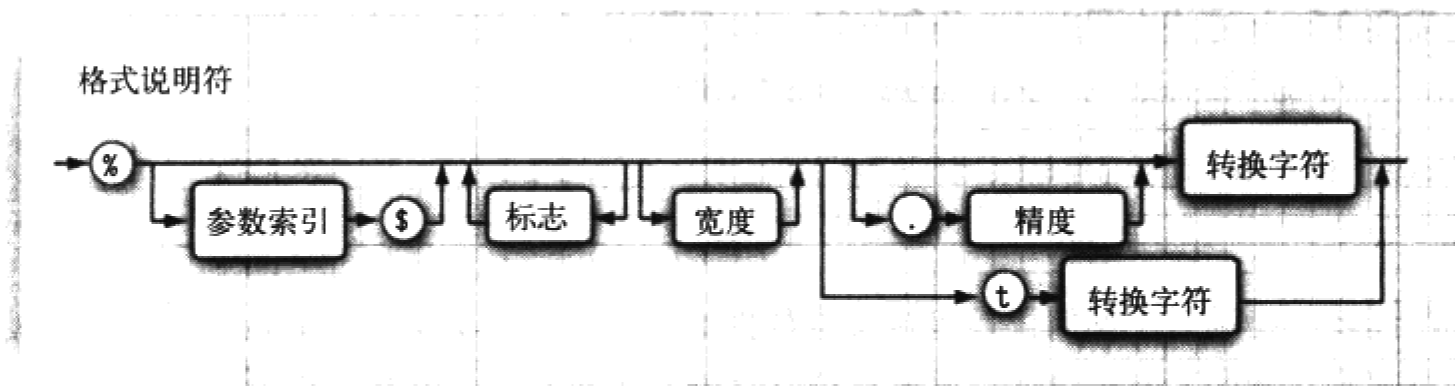


图3-6 格式说明符语法

3.7.3 文件输入与输出

要想对文件进行读取，就需要一个用File对象构造一个Scanner对象，如下所示：

```
Scanner in = new Scanner(new File("myfile.txt"));
```

如果文件名中包含反斜杠符号，就要记住在每个反斜杠之前再加一个额外的反斜杠："c:\\mydirectory\\myfile.txt"。

现在，就可以利用前面介绍的任何一个Scanner方法对文件进行读取。

要想写入文件，就需要构造一个PrintWriter对象。在构造器中，只需要提供文件名：

```
PrintWriter out = new PrintWriter("myfile.txt");
```

如果文件不存在，则可以像输出到System.out一样使用print、println以及printf命令。

警告：可以构造一个带有字符串参数的Scanner，但这个Scanner将字符串解释为数据，而不是文件名。例如，如果调用：

```
Scanner in = new Scanner("myfile.txt"); // ERROR?
```

这个scanner会将参数作为包含10个字符的数据：'m'，'y'，'f'等等。在这个示例中所显示的并不是人们所期望的效果。

注释：当指定一个相对文件名时，例如，"myfile.txt"，"mydirectory/myfile.txt"或"../myfile.txt"，文件位于Java虚拟机启动路径的相对位置。如果在命令行方式下用下列命令启动程序：

```
java MyProg
```

启动路径就是命令解释器的当前路径。然而，如果使用集成开发环境，那么启动路径将由IDE控制。可以使用下面的调用方式找到路径的位置：

```
String dir = System.getProperty("user.dir");
```

如果觉得定位文件比较烦恼，则可以考虑使用绝对路径，例如："c:\\mydirectory\\myfile.txt"或者"/home/me/mydirectory/myfile.txt"。

正如读者所看到的，访问文件与使用System.in和System.out一样容易。要记住一点：如果用一个不存在的文件构造一个Scanner，或者用一个不能被创建的文件名构造一个PrintWriter，那么就会发生异常。Java编译器认为这些异常比"被零整除"异常更严重。在第11章中，将会学习各种处理异常的方式。现在，应该告知编译器：已经知道有可能出现"找不到文件"的异

常。这需要在main方法中用throws子句标记，如下所示：

```
public static void main(String[] args) throws FileNotFoundException
{
   Scanner in = new Scanner(new File("myfile.txt"));
   . . .
}
```

现在读者已经学习了如何读写包含文本数据的文件。对于更加高级的技术，例如，处理不同的字符编码、处理二进制数据、读取目录以及编写压缩文件，请参看卷II第1章。

注释：当采用命令行方式启动一个程序时，可以利用重定向将任意文件捆绑到System.in和System.out:

```
java MyProg < myfile.txt > output.txt
```

这样，就不必担心处理FileNotFoundException异常了。

API java.util.Scanner 5.0

- `Scanner(File f)`

构造一个从给定文件读取数据的Scanner。

- `Scanner(String data)`

构造一个从给定字符串读取数据的Scanner。

API java.io.PrintWriter 1.1

- `PrintWriter(File f)`

构造一个将数据写入给定文件的PrintWriter。

- `PrintWriter(String fileName)`

构造一个将数据写入文件的PrintWriter。文件名由参数指定。

API java.io.File 1.0

- `File(String fileName)`

用给定文件名，构造一个描述文件的File对象。注意这个文件当前不必存在。

3.8 控制流程

与任何程序设计语言一样，Java使用条件语句和循环结构确定控制流程。本节先讨论条件语句，然后讨论循环语句，最后介绍看似有些笨重的switch语句，当需要对某个表达式的多个值进行检测时，可以使用switch语句。

C++注释：Java的控制流程结构与C和C++的控制流程结构一样，只有很少的例外情况。没有goto语句，但break语句可以带标签，可以利用它实现从内层循环跳出的目的（这种情况C语言采用goto语句实现）。另外，Java SE 5.0还添加了一种变形的for循环，在C或C++中没有这类循环。它有点类似于C#中的foreach循环。

3.8.1 块作用域

在深入学习控制结构之前，需要了解块（block）的概念。

块（即复合语句）是指由一对花括号括起来的若干条简单的Java语句。块确定了变量的作用域。一个块可以嵌套在另一个块中。下面就是在main方法块中嵌套另一个语句块的示例。

```
public static void main(String[] args)
{
   int n;
   . . .
   {
      int k;
      . . .
   } // k is only defined up to here
}
```

但是，不能在嵌套的两个块中声明同名的变量。例如，下面的代码就有错误，而无法通过编译：

```
public static void main(String[] args)
{
   int n;
   . . .
   {
      int k;
      int n; // error--can't redefine n in inner block
      . . .
   }
}
```

C++注释：在C++中，可以在嵌套的块中重定义一个变量。在内层定义的变量会覆盖在外层定义的变量。这样，有可能会导致程序设计错误，因此在Java中不允许这样做。

3.8.2 条件语句

在Java中，条件语句的格式为

if (*condition*) *statement*

这里的条件必须用括号括起来。

与绝大多数程序设计语言一样，Java常常希望在某个条件为真时执行多条语句。在这种情况下，应该使用块语句（block statement），格式为

```
{
   statement1
   statement2
   . . .
}
```

例如：

```
if (yourSales >= target)
{
   performance = "Satisfactory";
   bonus = 100;
}
```

当yourSales大于或等于target时，将执行括号中的所有语句（请参看图3-7）。

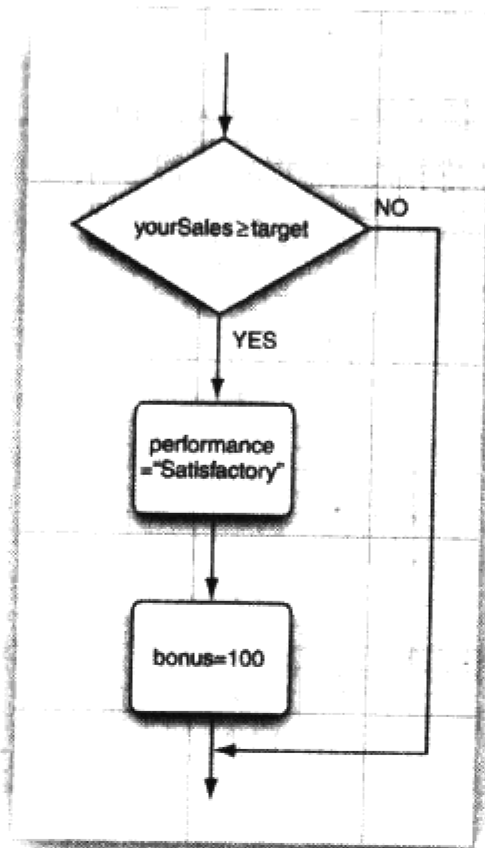


图3-7 if语句的流程图

✔ **注释**：使用块（有时称为复合语句）可以在Java程序结构中原本只能放置一条简单语句的地方放置多条语句。

在Java中，比较常见的条件语句格式如下所示（请参看图3-8）：

if (*condition*) *statement*$_1$ else *statement*$_2$

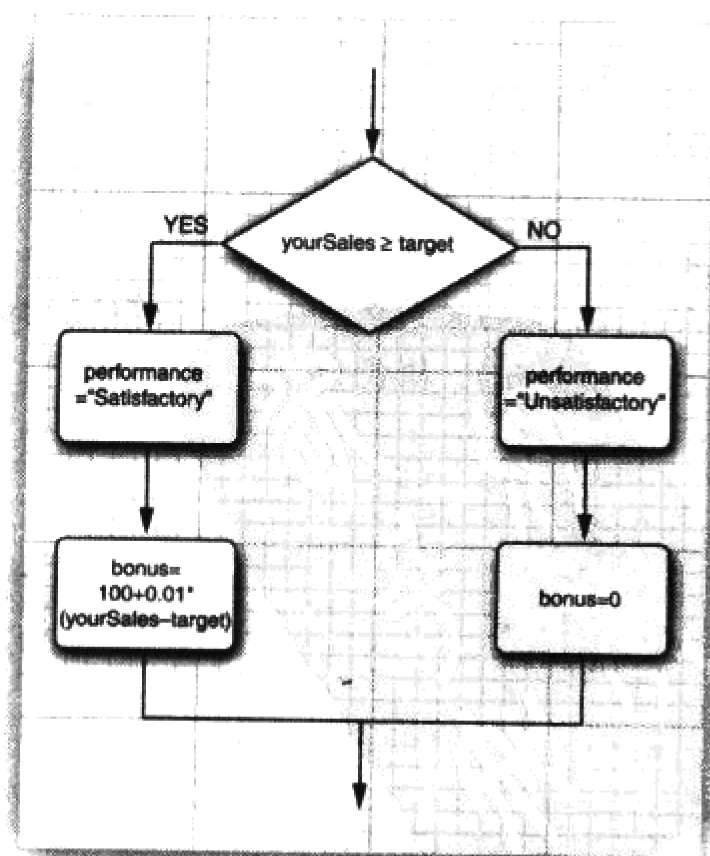


图3-8 if/else语句的流程图

例如：

```
if (yourSales >= target)
{
   performance = "Satisfactory";
   bonus = 100 + 0.01 * (yourSales - target);
}
else
{
   performance = "Unsatisfactory";
   bonus = 0;
}
```

其中else部分是可选的。else子句与最邻近的if构成一组。因此，在语句

```
if (x <= 0) if (x == 0) sign = 0; else sign = -1;
```

中else与第2个if配对。当然，用一对括号将会使这段代码更加清晰：

```
if (x <= 0) { if (x == 0) sign = 0; else sign = -1; }
```

重复地交替出现if...else if...是一种很常见的情况（请参看图3-9）。例如：

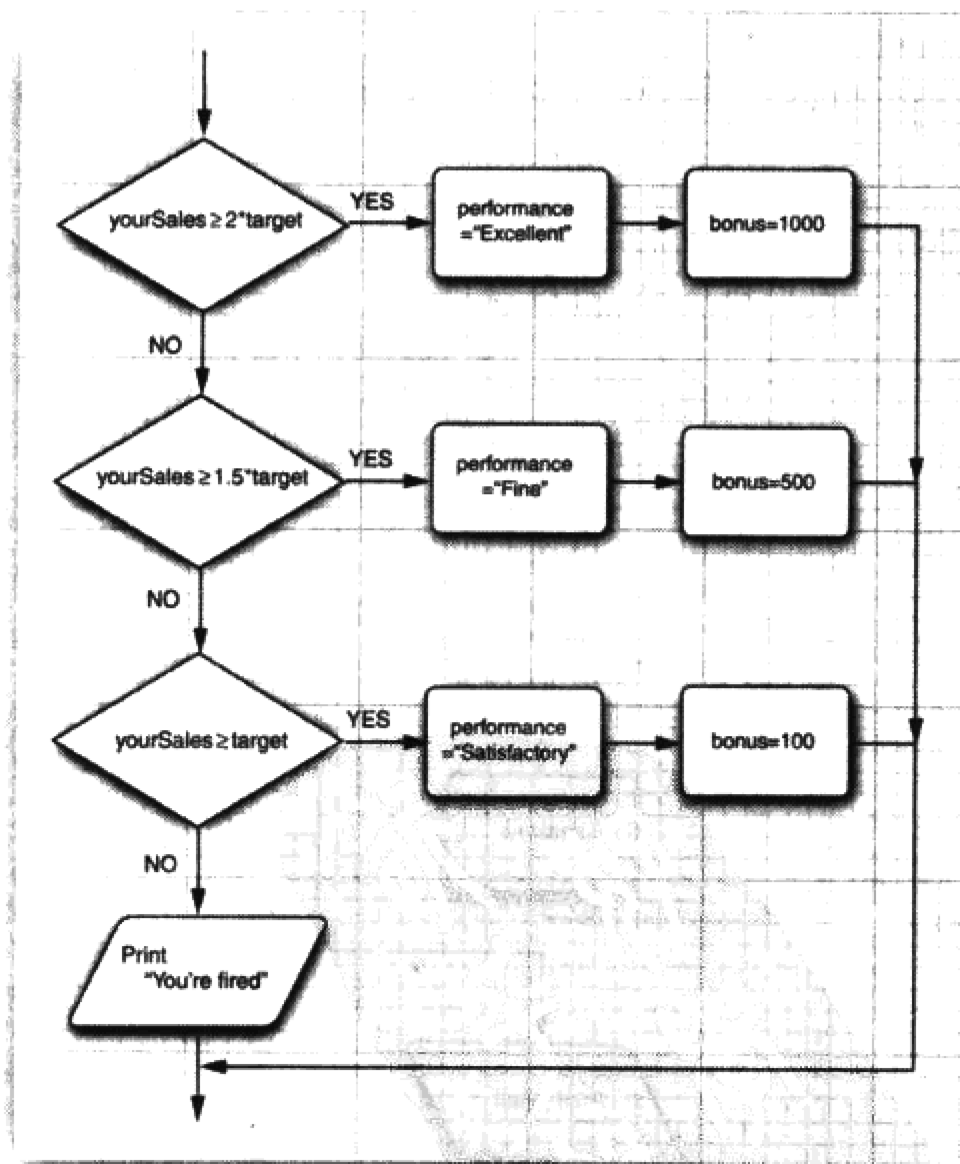


图3-9 if/else if（多分支）的流程图

```
if (yourSales >= 2 * target)
{
   performance = "Excellent";
```

```
    bonus = 1000;
}
else if (yourSales >= 1.5 * target)
{
    performance = "Fine";
    bonus = 500;
}
else if (yourSales >= target)
{
    performance = "Satisfactory";
    bonus = 100;
}
else
{
    System.out.println("You're fired");
}
```

3.8.3 循环

当条件为true时，while循环执行一条语句（也可以是一个语句块）。常用的格式为

while (*condition*) *statement*

如果开始循环条件的值就为false，则while循环体一次也不执行（请参看图3-10）。

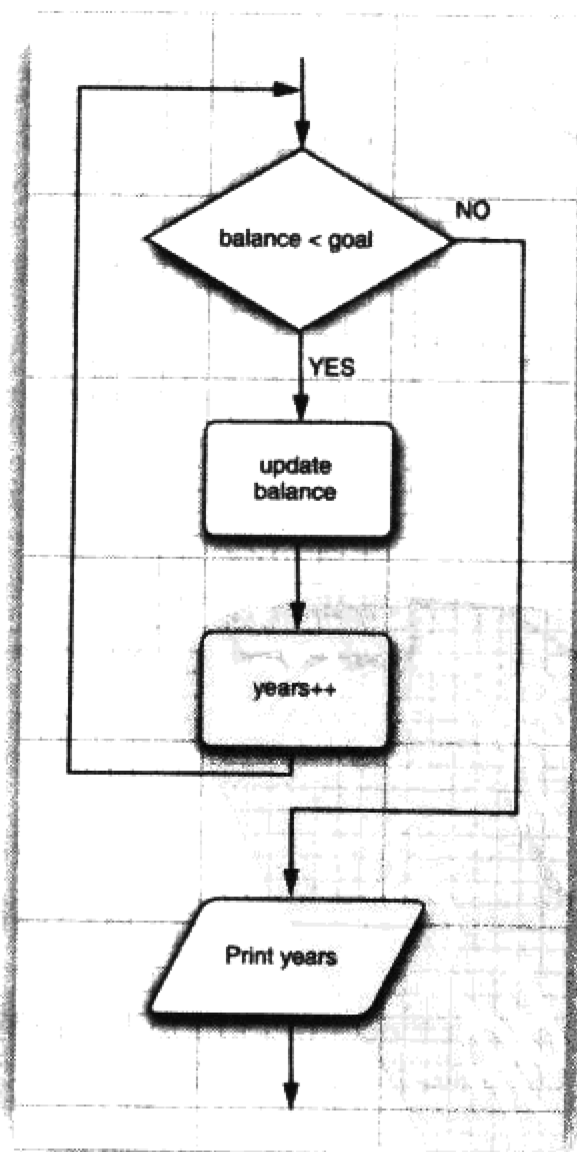


图3-10 while语句的流程图

例3-3的程序将计算需要多长时间才能够存储一定数量的退休金，假定每年存入相同数量的金额，而且利率是固定的。

在这个示例中，增加了一个计数器，并在循环体中更新当前的累积数量，直到总值超过目标值为止。

```
while (balance < goal)
{
   balance += payment;
   double interest = balance * interestRate / 100;
   balance += interest;
   years++;
}
System.out.println(years + " years.");
```

（千万不要使用这个程序安排退休计划。这里忽略了通货膨胀和所期望的生活水准。）

while循环语句首先检测循环条件。因此，循环体中的代码有可能不被执行。如果希望循环体至少执行一次，则应该将检测条件放在最后。使用do/while循环语句可以实现这种操作方式。它的语法格式为：

```
do statement while (condition);
```

这种循环语句先执行语句（通常是一个语句块），再检测循环条件；然后重复语句，再检测循环条件，以此类推。在例3-4的代码中，首先计算退休账户中的余额，然后再询问是否打算退休：

```
do
{
   balance += payment;
   double interest = balance * interestRate / 100;
   balance += interest;
   year++;
   // print current balance
   . . .
   // ask if ready to retire and get input
   . . .
}
while (input.equals("N"));
```

只要用户回答“N”，循环就重复执行（见图3-11）。这是一个需要至少执行一次的循环的很好示例，因为用户必须先看到余额才能知道是否满足退休所用。

例3-3 Retirement.java

```
import java.util.*;

/**
 * This program demonstrates a <code>while</code> loop.
 * @version 1.20 2004-02-10
 * @author Cay Horstmann
 */
public class Retirement
{
   public static void main(String[] args)
   {
      // read inputs
      Scanner in = new Scanner(System.in);
```

```

        System.out.print("How much money do you need to retire? ");
        double goal = in.nextDouble();

        System.out.print("How much money will you contribute every year? ");
        double payment = in.nextDouble();

        System.out.print("Interest rate in %: ");
        double interestRate = in.nextDouble();

        double balance = 0;
        int years = 0;

        // update account balance while goal isn't reached
        while (balance < goal)
        {
           // add this year's payment and interest
           balance += payment;
           double interest = balance * interestRate / 100;
           balance += interest;
           years++;
        }

        System.out.println("You can retire in " + years + " years.");
     }
  }
```

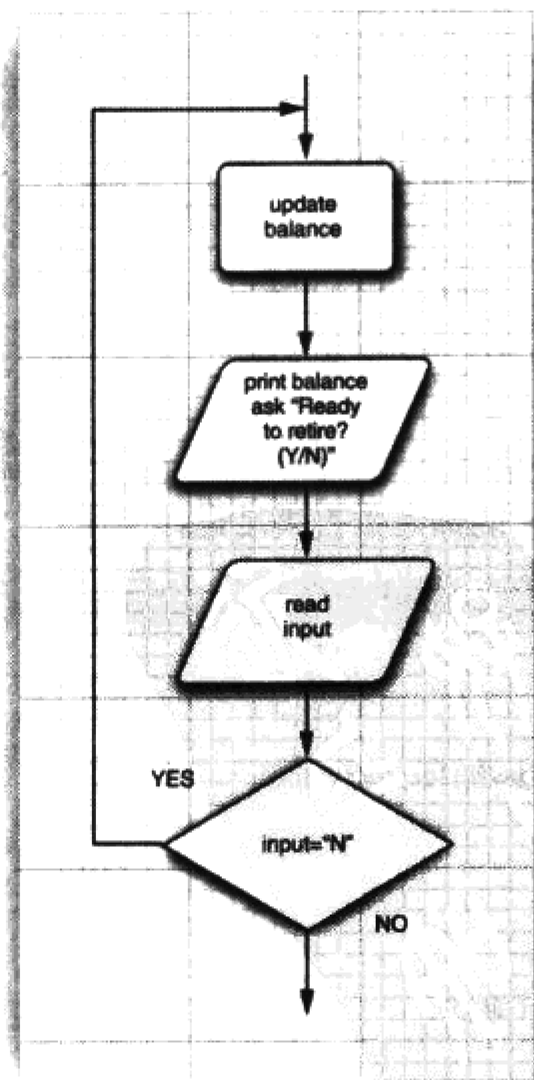


图3-11 do/while语句的流程图

例3-4 Retirement2.java

```
import java.util.*;

/**
 * This program demonstrates a <code>do/while</code> loop.
 * @version 1.20 2004-02-10
 * @author Cay Horstmann
 */
public class Retirement2
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);

      System.out.print("How much money will you contribute every year? ");
      double payment = in.nextDouble();

      System.out.print("Interest rate in %: ");
      double interestRate = in.nextDouble();

      double balance = 0;
      int year = 0;

      String input;

      // update account balance while user isn't ready to retire
      do
      {
         // add this year's payment and interest
         balance += payment;
         double interest = balance * interestRate / 100;
         balance += interest;

         year++;

         // print current balance
         System.out.printf("After year %d, your balance is %,.2f%n", year, balance);

         // ask if ready to retire and get input
         System.out.print("Ready to retire? (Y/N) ");
         input = in.next();
      }
      while (input.equals("N"));
   }
}
```

3.8.4 确定循环

for循环语句是支持迭代的一种通用结构，利用每次迭代之后更新的计数器或类似的变量来控制迭代次数。如图3-12所示，下面的程序将数字1～10输出到屏幕上。

```
for (int i = 1; i <= 10; i++)
   System.out.println(i);
```

for语句的第1部分通常用于对计数器初始化；第2部分给出每次新一轮循环执行前要检测的

循环条件；第3部分指示如何更新计数器。

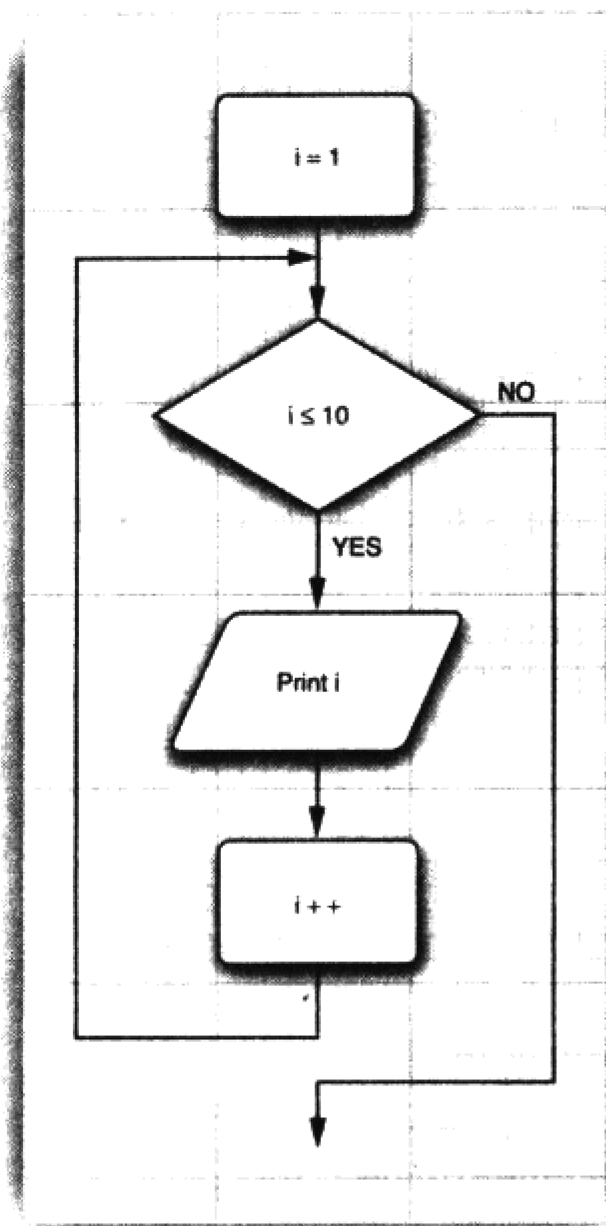


图3-12 for语句的流程图

与C++一样，尽管Java允许在for循环的各个部分放置任何表达式，但有一条不成文的规则：for语句的3个部分应该对同一个计数器变量进行初始化、检测和更新。若不遵守这一规则，编写的循环常常晦涩难懂。

即使遵守了这条规则，也还有可能出现很多问题。例如，下面这个倒计数的循环：

```
for (int i = 10; i > 0; i--)
   System.out.println("Counting down . . . " + i);
System.out.println("Blastoff!");
```

警告：在循环中，检测两个浮点数是否相等需要格外小心。下面的for循环

```
for (double x = 0; x != 10; x += 0.1) . . .
```

可能永远不会结束。由于舍入的误差，最终可能得不到精确值。例如，在上面的循环中，因为0.1无法精确地用二进制表示，所以，x将从9.999 999 999 999 98跳到10.099 999 999 999 98。

当在for语句的第1部分中声明了一个变量之后，这个变量的作用域就为for循环的整个循环体。

```
for (int i = 1; i <= 10; i++)
{
   . . .
}
// i no longer defined here
```

特别指出，如果在for语句内部定义一个变量，这个变量就不能在循环体之外使用。因此，如果希望在for循环体之外使用循环计数器的最终值，就要确保这个变量在循环语句的前面且在外部声明！

```
int i;
for (i = 1; i <= 10; i++)
{
   . . .
}
// i still defined here
```

另一方面，可以在各自独立的不同for循环中定义同名的变量：

```
for (int i = 1; i <= 10; i++)
{
   . . .
}
. . .
for (int i = 11; i <= 20; i++) // ok to define another variable named i
{
   . . .
}
```

for循环语句只不过是while循环的一种简化形式。例如，

```
for (int i = 10; i > 0; i--)
   System.out.println("Counting down . . . " + i);
```

可以重写为：

```
int i = 10;
while (i > 0)
{
   System.out.println("Counting down . . . " + i);
   i--;
}
```

例3-5给出了一个应用for循环的典型示例。这个程序用来计算抽奖中奖的概率。例如，如果必须从1～50之间的数字中取6个数字来抽奖，那么会有 $(50 \times 49 \times 48 \times 47 \times 46 \times 45)/(1 \times 2 \times 3 \times 4 \times 5 \times 6)$ 种可能的结果，所以中奖的几率是1/15 890 700。祝你好运！

一般情况下，如果从n个数字中抽取k个数字，就可以使用下列公式得到结果。

$$\frac{n \times (n-1) \times (n-2) \times \cdots \times (n-k+1)}{1 \times 2 \times 3 \times \cdots \times k}$$

下面的for循环语句计算了上面这个公式的值：

```
int lotteryOdds = 1;
for (int i = 1; i <= k; i++)
   lotteryOdds = lotteryOdds * (n - i + 1) / i;
```

注释：稍后将会介绍"通用for循环"（又称为for each循环），这是Java SE 5.0新增加的一种循环结构。

例3-5 LotteryOdds.java

```
import java.util.*;

/**
 * This program demonstrates a <code>for</code> loop.
 * @version 1.20 2004-02-10
 * @author Cay Horstmann
 */
public class LotteryOdds
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);

      System.out.print("How many numbers do you need to draw? ");
      int k = in.nextInt();

      System.out.print("What is the highest number you can draw? ");
      int n = in.nextInt();

      /*
       * compute binomial coefficient n*(n-1)*(n-2)*...*(n-k+1)/(1*2*3*...*k)
       */

      int lotteryOdds = 1;
      for (int i = 1; i <= k; i++)
         lotteryOdds = lotteryOdds * (n - i + 1) / i;

      System.out.println("Your odds are 1 in " + lotteryOdds + ". Good luck!");
   }
}
```

3.8.5 多重选择：switch语句

在处理多个选项时，使用if/else结构显得有些笨拙。Java有一个与C/C++完全一样的switch语句。

例如，如果建立一个如图3-13所示的包含4个选项的菜单系统，就应该使用下列代码：

```
Scanner in = new Scanner(System.in);
System.out.print("Select an option (1, 2, 3, 4) ");
int choice = in.nextInt();
switch (choice)
{
   case 1:
      . . .
      break;
   case 2:
      . . .
      break;
   case 3:
      . . .
```

```
         break;
      case 4:
         . . .
         break;
      default:
         // bad input
         . . .
         break;
   }
```

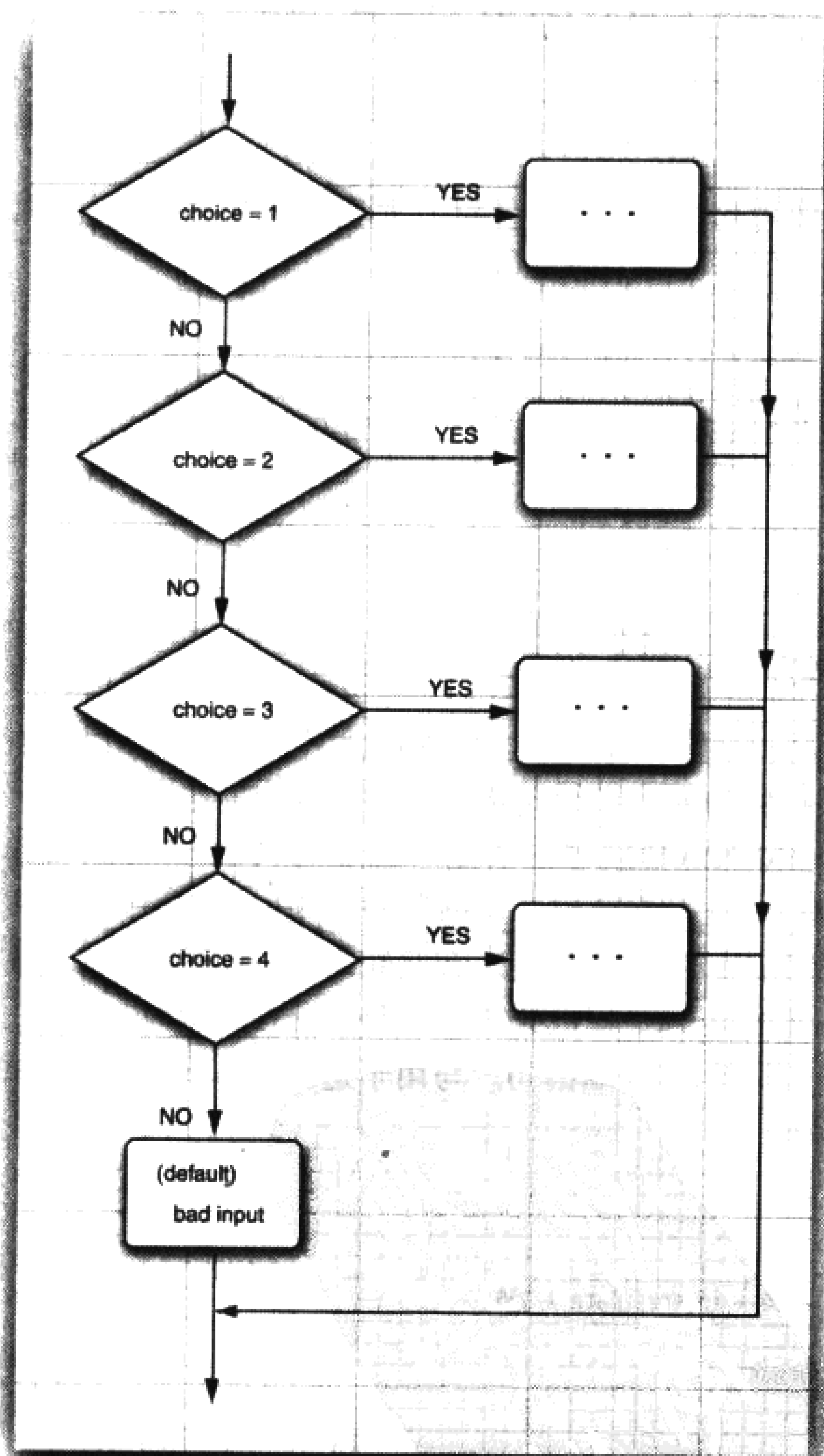


图3-13 switch语句的流程图

switch语句将从与选项值相匹配的case标签处开始执行直到遇到break语句，或者执行到switch语句的结束处为止。如果没有相匹配的case标签，而有default子句，就执行这个子句。

警告：有可能触发多个case分支。如果在case分支语句的末尾没有break语句，那么就会接着执行下一个case分支语句。这种情况相当危险，常常会引发错误。为此，我们在程序中从不使用switch语句。

case标签必须是整数或枚举常量，不能测试字符串。例如，下面这段代码就存在错误。

```
String input = . . .;
switch (input) // ERROR
{
   case "A": // ERROR
      . . .
      break;
   . . .
}
```

当在switch语句中使用枚举常量时，不必在每个标签中指明枚举名，可以由switch的表达式值确定。例如：

```
Size sz = . . .;
switch (sz)
{
   case SMALL: // no need to use Size.SMALL
      . . .
      break;
   . . .
}
```

3.8.6 中断控制流程语句

尽管Java的设计者将goto作为保留字，但实际上并没有打算在语言中使用它。通常，使用goto语句被认为是一种拙劣的程序设计风格。当然，也有一些程序员认为反对goto的呼声似乎有些过分（例如，Donald Knuth就曾编著过一篇名为《Structured Programming with goto statements》的著名文章）。这篇文章说：无限制地使用goto语句确实是导致错误的根源，但在有些情况下，偶尔地使用goto跳出循环还是有益处的。Java设计者同意这种看法，甚至在Java语言中增加了一条带标签的break，以此来支持这种程序设计风格。

下面首先看一下不带标签的break语句。与用于退出switch语句的break语句一样，它也可以用于退出循环语句。例如，

```
while (years <= 100)
{
   balance += payment;
   double interest = balance * interestRate / 100;
   balance += interest;
   if (balance >= goal) break;
   years++;
}
```

在循环开始时，如果years > 100，或者在循环体中balance≥goal，则退出循环语句。当然，也可以在不使用break的情况下计算years的值，如下所示：

```
while (years <= 100 && balance < goal)
{
```

```
   balance += payment;
   double interest = balance * interestRate / 100;
   balance += interest;
   if (balance < goal)
      years++;
}
```

但是需要注意，在这个版本中，检测了两次balance < goal。为了避免重复检测，有些程序员更加偏爱使用break语句。

与C++不同，Java还提供了一种带标签的break语句，用于跳出多重嵌套的循环语句。有时候，在嵌套很深的循环语句中会发生一些不可预料的事情。此时可能更加希望跳到嵌套的所有循环语句之外。通过添加一些额外的条件判断实现各层循环的检测很不方便。

这里有一个示例说明了break语句的工作状态。请注意，标签必须放在希望跳出的最外层循环之前，并且必须紧跟一个冒号。

```
Scanner in = new Scanner(System.in);
int n;
read_data:
while (. . .) // this loop statement is tagged with the label
{
   . . .
   for (. . .) // this inner loop is not labeled
   {
      System.out.print("Enter a number >= 0: ");
      n = in.nextInt();
      if (n < 0) // should never happen-can't go on
         break read_data;
         // break out of read_data loop
      . . .
   }
}
// this statement is executed immediately after the labeled break
if (n < 0) // check for bad situation
{
   // deal with bad situation
}
else
{
   // carry out normal processing
}
```

如果输入有误，通过执行带标签的break跳转到带标签的语句块末尾。对于任何使用break语句的代码都需要检测循环是正常结束，还是由break跳出。

注释：事实上，可以将标签应用到任何语句中，甚至可以应用到if语句或者块语句中，如下所示：

```
label:
{
   . . .
   if (condition) break label; // exits block
   . . .
}
// jumps here when the break statement executes
```

因此，如果希望使用一条goto语句，并将一个标签放在想要跳到的语句块之前，就可以使用break语句！当然，并不提倡使用这种方式。另外需要注意，只能跳出语句块，而不能跳入语句块。

最后，还有一个continue语句。与break语句一样，它将中断正常的控制流程。continue语句将控制转移到最内层循环的首部。例如：

```
Scanner in = new Scanner(System.in);
while (sum < goal)
{
   System.out.print("Enter a number: ");
   n = in.nextInt();
   if (n < 0) continue;
   sum += n; // not executed if n < 0
}
```

如果n<0，则continue语句越过了当前循环体的剩余部分，立刻跳到循环首部。

如果将continue语句用于for循环中，就可以跳到for循环的“更新”部分。例如，一下这个循环：

```
for (count = 1; count <= 100; count++)
{
   System.out.print("Enter a number, -1 to quit: ");
   n = in.nextInt();
   if (n < 0) continue;
   sum += n; // not executed if n < 0
}
```

如果n<0，则continue语句跳到count++语句。

还有一种带标签的continue语句，将跳到与标签匹配的循环首部。

提示：许多程序员容易混淆break和continue语句。这些语句完全是可选的，即不使用它们也可以表达同样的逻辑含义。在本书中，将不使用break和continue。

3.9 大数值

如果基本的整数和浮点数精度不能够满足需求，那么可以使用java.math包中的两个很有用的类：BigInteger和BigDecimal。这两个类可以处理包含任意长度数字序列的数值。BigInteger类实现了任意精度的整数运算，BigDecimal实现了任意精度的浮点数运算。

使用静态的valueOf方法可以将普通的数值转换为大数值：

```
BigInteger a = BigInteger.valueOf(100);
```

遗憾的是，不能使用人们熟悉的算术运算符（如：+和*）处理大数值。而需要使用大数值类中的add和multiply方法。

```
BigInteger c = a.add(b); // c = a + b
BigInteger d = c.multiply(b.add(BigInteger.valueOf(2))); // d = c * (b + 2)
```

C++注释：与C++不同，Java没有提供运算符重载功能。程序员无法重定义+和*运算符，使其应用于BigInteger类的add和multiply运算。Java语言的设计者确实为字符串的连接重

载了+运算符，但没有重载其他的运算符，也没有给Java程序员自己重载运算符的权利。

例3-6是对例3-5中彩概率程序的改进，使其可以采用大数值进行运算。假设你被邀请参加抽奖活动，并从490个可能的数值中抽取60个，这个程序将会得到中彩概率1/716395843461995557415116222540092933411717612789263493493351013459481104668848。祝你好运！

在例3-5程序中，用于计算的语句是

```
lotteryOdds = lotteryOdds * (n - i + 1) / i;
```

如果使用大数值，则相应的语句为：

```
lotteryOdds = lotteryOdds.multiply(BigInteger.valueOf(n - i + 1)).divide(BigInteger.valueOf(i));
```

例3-6 BigIntegerTest.java

```
import java.math.*;
import java.util.*;

/**
 * This program uses big numbers to compute the odds of winning the grand prize in a lottery.
 * @version 1.20 2004-02-10
 * @author Cay Horstmann
 */
public class BigIntegerTest
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);

      System.out.print("How many numbers do you need to draw? ");
      int k = in.nextInt();

      System.out.print("What is the highest number you can draw? ");
      int n = in.nextInt();

      /*
       * compute binomial coefficient n*(n-1)*(n-2)*...*(n-k+1)/(1*2*3*...*k)
       */

      BigInteger lotteryOdds = BigInteger.valueOf(1);

      for (int i = 1; i <= k; i++)
         lotteryOdds = lotteryOdds.multiply(BigInteger.valueOf(n - i + 1)).divide(
               BigInteger.valueOf(i));

      System.out.println("Your odds are 1 in " + lotteryOdds + ". Good luck!");
   }
}
```

API java.math.BigInteger 1.1

- `BigInteger add(BigInteger other)`
- `BigInteger subtract(BigInteger other)`

- `BigInteger multiply(BigInteger other)`
- `BigInteger divide(BigInteger other)`
- `BigInteger mod(BigInteger other)`

  返回这个大整数和另一个大整数other的和、差、积、商以及余数。
- `int compareTo(BigInteger other)`

  如果这个大整数与另一个大整数other相等，返回0；如果这个大整数小于另一个大整数other，返回负数；否则，返回正数。
- `static BigInteger valueOf(long x)`

  返回值等于x的大整数。

API java.math.BigInteger 1.1

- `BigDecimal add(BigDecimal other)`
- `BigDecimal subtract(BigDecimal other)`
- `BigDecimal multiply(BigDecimal other)`
- `BigDecimal divide(BigDecimal other, RoundingMode mode) 5.0`

  返回这个大实数与另一个大实数other的和、差、积、商。要想计算商，必须给出舍入方式（rounding mode）。RoundingMode.HALF_UP是在学校中学习的四舍五入方式（即，数值0到4舍去，数值5到9进位）。它适用于常规的计算。有关其他的舍入方式请参看API文档。
- `int compareTo(BigDecimal other)`

  如果这个大实数与另一个大实数相等，返回0；如果这个大实数小于另一个大实数，返回负数；否则，返回正数。
- `static BigDecimal valueOf(long x)`
- `static BigDecimal valueOf(long x,int scale)`

  返回值为x或x / 10^{scale}的一个大实数。

3.10 数组

数组是一种数据结构，用来存储同一类型值的集合。通过一个整型下标可以访问数组中的每一个值。例如，如果a是一个整型数组，a[i]就是数组中下标为i的整数。

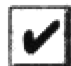
在声明数组变量时，需要指出数组类型（数据元素类型紧跟[]）和数组变量的名字。下面声明了整型数组a：

```
int[] a;
```

这条语句只声明了变量a，并没有将a初始化为一个真正的数组。应该使用new运算符创建数组。

```
int[] a = new int[100];
```

这条语句创建了一个可以存储100个整数的数组。



注释：可以使用下面两种形式声明数组

```
int[] a;
```

或

```
int a[];
```

大多数Java应用程序员喜欢使用第一种风格，因为它将类型int[]（整型数组）与变量名分开了。

这个数组的下标从0～99（不是1～100）。一旦创建了数组，就可以给数组元素赋值。例如，使用一个循环：

```
int[] a = new int[100];
for (int i = 0; i < 100; i++)
   a[i] = i;  // fills the array with 0 to 99
```

警告：如果创建了一个100个元素的数组，并且试图访问元素a[100]（或任何在0～99之外的下标），程序就会引发“array index out of bounds”异常而终止执行。

要想获得数组中的元素个数，可以使用array.length。例如，

```
for (int i = 0; i < a.length; i++)
   System.out.println(a[i]);
```

一旦创建了数组，就不能再改变它的大小（尽管可以改变每一个数组元素）。如果经常需要在运行过程中扩展数组的大小，就应该使用另一种数据结构——数组列表（array list）有关数组列标的详细内容请参看第5章。

3.10.1 For each循环

Java SE 5.0增加了一种功能很强的循环结构，可以用来依次处理数组中的每个元素（其他类型的元素集合亦可）而不必为指定下标值而分心。

这种增强的for循环的语句格式为：

for (*variable* : *collection*) *statement*

定义一个变量用于暂存集合中的每一个元素，并执行相应的语句（当然，也可以是语句块）。collection这一集合表达式必须是一个数组或者是一个实现了Iterable接口的类对象（例如ArrayList）。有关数组列表的内容将在第5章中讨论，有关Iterable接口的内容将在卷II的第2章中讨论。

例如，

```
for (int element : a)
   System.out.println(element);
```

打印数组a的每一个元素，一个元素占一行。

这个循环应该读作“循环a中的每一个元素”（for each element in a）。Java语言的设计者认为应该使用诸如foreach、in这样的关键字，但这种循环语句并不是最初就包含在Java语言中的，而是后来添加进去的，并且没有人打算废除已经包含同名（例如System.in）方法或变量的旧代码。

当然，使用传统的for循环也可以获得同样的效果：

```
for (int i = 0; i < a.length; i++)
   System.out.println(a[i]);
```

但是，for each循环语句显得更加简洁、更不易出错（不必为下标的起始值和终止值而操心）。

注释：for each循环语句的循环变量将会遍历数组中的每个元素，而不需要使用下标值。

如果需要处理一个集合中的所有元素，for each循环语句对传统循环语句所进行的改进更是叫人称赞不已。然而，在很多场合下，还是需要使用传统的for循环。例如，如果不希望遍历集合中的每个元素，或者在循环内部需要使用下标值等。

提示：有个更加简单的方式打印数组中的所有值，即利用Arrays类的toString方法。调用Arrays.toString(a)，返回一个包含数组元素的字符串，这些元素被放置在括号内，并用逗号分隔，例如，"[2,3,5,7,11,13]"。要想打印数组，可以调用

```
System.out.println(Arrays.toString(a));
```

3.10.2 数组初始化以及匿名数组

在Java中，提供了一种创建数组对象并同时赋予初始值的简化书写形式。下面是一个例子：

```
int[] smallPrimes = { 2, 3, 5, 7, 11, 13 };
```

请注意，在使用这种语句时，不需要调用new。

甚至还可以初始化一个匿名的数组：

```
new int[] { 17, 19, 23, 29, 31, 37 }
```

这种表示法将创建一个新数组并利用括号中提供的值进行初始化，数组的大小就是初始值的个数。使用这种语法形式可以在不创建新变量的情况下重新初始化一个数组。例如：

```
smallPrimes = new int[] { 17, 19, 23, 29, 31, 37 };
```

这是下列语句的简写形式：

```
int[] anonymous = { 17, 19, 23, 29, 31, 37 };
smallPrimes = anonymous;
```

注释：在Java中，允许数组长度为0。在编写一个结果为数组的方法时，如果碰巧结果为空，则这种语法形式就显得非常有用。此时可以创建一个长度为0的数组：

```
new elementType[0]
```

注意，数组长度为0与null不同（关于null的详细论述请参看第4章）。

3.10.3 数组拷贝

在Java中，允许将一个数组变量拷贝给另一个数组变量。这时，两个变量将引用同一个数组：

```
int[] luckyNumbers = smallPrimes;
luckyNumbers[5] = 12; // now smallPrimes[5] is also 12
```

图3-14显示了拷贝的结果。如果希望将一个数组的所有值拷贝到一个新的数组中去，就要使用Arrays类的copyOf方法：

```
int[] copiedLuckyNumbers = Arrays.copyOf(luckyNumbers, luckyNumbers.length);
```

第2个参数是新数组的长度。这个方法通常用来增加数组的大小：

```
luckyNumbers = Arrays.copyOf(luckyNumbers, 2 * luckyNumbers.length);
```

如果数组元素是数值型，那么多余的元素将被赋值为0；如果数组元素是布尔型，则将赋值为

false。相反，如果长度小于原始数组的长度，则只拷贝最前面的数据元素。

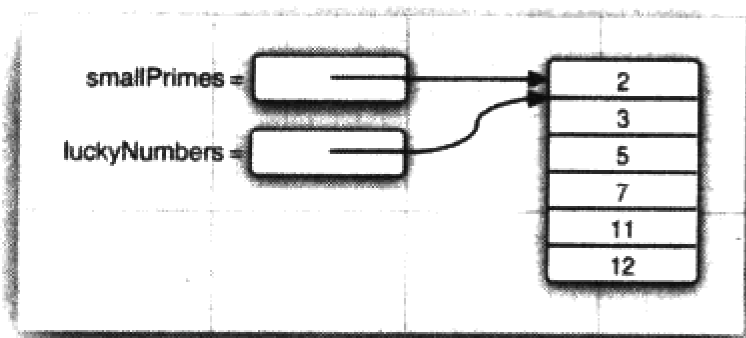


图3-14 拷贝一个数组变量

数组to必须有足够的空间存放拷贝的元素。

注释：在Java SE 6之前，用System类的arraycopy方法将一个数组的元素拷贝到另一个数组中。调用这个方法的语法格式为：

```
System.arraycopy(from, fromIndex, to, toIndex, count);
```

数组to必须有足够的空间存放拷贝的元素。

例如，下面这段语句所得到的结果如图3-15所示。它创建了两个数组，然后将第一个数组的后4个元素拷贝到第二个数组中。拷贝从原始数组的第2个位置开始，一共拷贝4个元素，目标数组的起始位置为3。

```
int[] smallPrimes = {2, 3, 5, 7, 11, 13};
int[] luckyNumbers = {1001, 1002, 1003, 1004, 1005, 1006, 1007};
System.arraycopy(smallPrimes, 2, luckyNumbers, 3, 4);
for (int i = 0; i < luckyNumbers.length; i++)
   System.out.println(i + ": " + luckyNumbers[i]);
```

输出结果为

```
0: 1001
1: 1002
2: 1003
3: 5
4: 7
5: 11
6: 13
```

C++注释：Java数组与C++数组在堆栈上有很大不同，但基本上与分配在堆（heap）上的数组指针一样。也就是说，

```
int[] a = new int[100]; // Java
```

不同于

```
int a[100]; // C++
```

而等同于

```
int* a = new int[100]; // C++
```

Java中的[ ]运算符被预定义为检查数组边界，而且没有指针运算，即不能通过a加1得到数组的下一个元素。

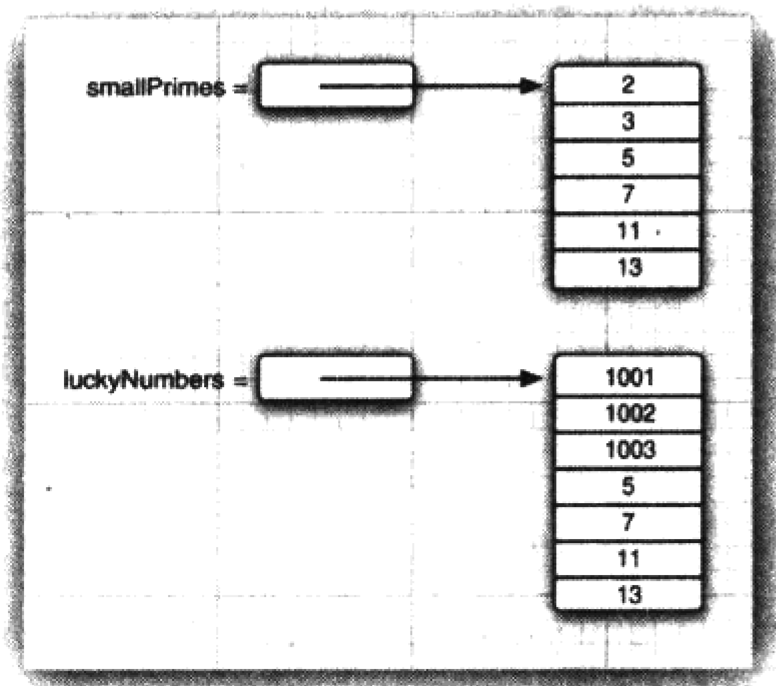


图3-15 数组之间拷贝元素值

3.10.4 命令行参数

前面已经看到多个使用Java数组的示例。每一个Java应用程序都有一个带String arg[]参数的main方法。这个参数表明main方法将接收一个字符串数组，也就是命令行参数。

例如，看一看下面这个程序：

```
public class Message
{
   public static void main(String[] args)
   {
      if (args[0].equals("-h"))
         System.out.print("Hello,");
      else if (args[0].equals("-g"))
         System.out.print("Goodbye,");
      // print the other command-line arguments
      for (int i = 1; i < args.length; i++)
         System.out.print(" " + args[i]);
      System.out.println("!");
   }
}
```

如果使用下面这种形式运行这个程序：

```
java Message -g cruel world
```

args数组将包含下列内容：

```
args[0]: "-g"
args[1]: "cruel"
args[2]: "world"
```

这个程序将显示下列信息：

```
Goodbye, cruel world!
```


C++注释：在Java应用程序的main方法中，程序名并没有存储在args数组中。例如，当使用下列命令运行程序时

```
java Message -h world
```

args[0]是"-h"，而不是"Message"或"java"。

3.10.5 数组排序

要想对数值型数组进行排序，可以使用Arrays类中的sort方法：

```
int[] a = new int[10000];
. . .
Arrays.sort(a)
```

这个方法使用了优化的快速排序算法。快速排序算法对于大多数数据集合来说都是效率比较高的。Array类还提供了几个使用很便捷的方法，在稍后的API注释中将介绍它们。

例3-7中的程序用到了数组，它产生一个抽彩游戏中的随机数值组合。假如抽彩是从49个数值中抽取6个，那么程序可能的输出结果为：

```
Bet the following combination. It'll make you rich!
  4
  7
  8
  19
  30
  44
```

要想选择这样一个随机的数值集合，就要首先将数值1，2，…，n存入数组numbers中：

```
int[] numbers = new int[n];
for (int i = 0; i < numbers.length; i++)
   numbers[i] = i + 1;
```

而用第二个数组存放抽取出来的数值：

```
int[] result = new int[k];
```

现在，就可以开始抽取k个数值了。Math.random方法将返回一个0到1之间（包含0、不包含1）的随机浮点数。用n乘以这个浮点数，就可以得到从0到n－1之间的一个随机数。

```
int r = (int) (Math.random() * n);
```

下面将result的第i个元素设置为numbers[r]存放的数值，最初是r+1。但正如所看到的，numbers数组的内容在每一次抽取之后都会发生变化。

```
result[i] = numbers[r];
```

现在，必须确保不会再次抽取到那个数值，因为所有抽彩的数值必须不相同。因此，这里用数组中的最后一个数值改写number[r]，并将n减1。

```
numbers[r] = numbers[n - 1];
n--;
```

关键在于每次抽取的都是下标，而不是实际的值。下标指向包含尚未抽取过的数组元素。

在抽取了k个数值之后，就可以对result数组进行排序了，这样可以让输出效果更加清晰：

```
Arrays.sort(result);
for (int r : result)
   System.out.println(r);
```

例3-7 LotteryDrawing.java

```
import java.util.*;

/**
 * This program demonstrates array manipulation.
 * @version 1.20 2004-02-10
 * @author Cay Horstmann
 */
public class LotteryDrawing
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);

      System.out.print("How many numbers do you need to draw? ");
      int k = in.nextInt();

      System.out.print("What is the highest number you can draw? ");
      int n = in.nextInt();

      // fill an array with numbers 1 2 3 . . . n
      int[] numbers = new int[n];
      for (int i = 0; i < numbers.length; i++)
         numbers[i] = i + 1;

      // draw k numbers and put them into a second array
      int[] result = new int[k];
      for (int i = 0; i < result.length; i++)
      {
         // make a random index between 0 and n - 1
         int r = (int) (Math.random() * n);

         // pick the element at the random location
         result[i] = numbers[r];

         // move the last element into the random location
         numbers[r] = numbers[n - 1];
         n--;
      }

      // print the sorted array
      Arrays.sort(result);
      System.out.println("Bet the following combination. It'll make you rich!");
      for (int r : result)
         System.out.println(r);
   }
}
```

API java.util.Arrays 1.2

- static String toString(type[] a) 5.0

  返回包含a中数据元素的字符串，这些数据元素被放在括号内，并用逗号分隔。

  参数：a　　类型为int、long、short、char、byte、boolean、float或double的数组。

- static type copyOf(*type*[] a, int length) 6
- static type copyOf(*type*[] a, int start, int end) 6

返回与a类型相同的一个数组，其长度为length或者 end-start，数组元素为a的值。

参数：a　　类型为int、long、short、char、byte、boolean、float或double的数组。

　　start　　起始下标（包含这个值）。

　　end　　终止下标（不包含这个值）。这个值可能大于a.length。在这种情况下，结果为0或false。

　　length　　拷贝的数据元素长度。如果length值大于a.length，结果为0或false；否则，数组中只有前面length个数据元素的拷贝值。

- static void sort(type[] a)

采用优化的快速排序算法对数组进行排序。

参数：a　　类型为int、long、short、char、byte、boolean、float或double的数组。

- static int binarySearch(*type*[] a, *type* v)
- static int binarySearch(*type*[] a, int start, int end, type v) 6

采用二分搜索算法查找值v。如果查找成功，则返回相应的下标值；否则，返回一个负数值r。－r－1是为保持a有序v应插入的位置。

参数：a　　类型为int、long、short、char、byte、boolean、float或double的有序数组。

　　start　　起始下标（包含这个值）。

　　end　　终止下标（不包含这个值）。

　　v　　同a的数据元素类型相同的值

- static void fill(*type*[] a, *type* v)

将数组的所有数据元素值设置为v

参数：a　　类型为int、long、short、char、byte、boolean、float或double的数组。

　　v　　与a数据元素类型相同的一个值。

- static boolean equals(type[] a, type[] b)

如果两个数组大小相同，并且下标相同的元素都对应相等，返回true。

参数：a、b　　类型为int、long、short、char、byte、boolean、float或double的两个数组。

API java.lang.System 1.1

- static void arraycopy(Object from, int fromIndex, Object to, int toIndex, int count)

将第一个数组中的元素拷贝到第二个数组中。

参数：from　　任意类型的数组（在第5章中将解释为什么这个参数的类型是Object）。

　　fromIndex　　原始数组中待拷贝元素的起始下标。

　　to　　与from同类型的数组。

　　toIndex　　目标数组放置拷贝元素的起始下标。

　　count　　拷贝的元素数量。

3.10.6 多维数组

多维数组将使用多个下标访问数组元素，它适用于表示表格或更加复杂的排列形式。这一节的内容可以先跳过，等到需要使用这种存储机制时再返回来学习。

假设需要建立一个数值表，用来显示在不同利率下投资 $10,000会增长多少，利息每年兑现，而且又被用于投资。表3-8是相应的图示。

表3-8 不同利率下的投资增长情况

10%	11%	12%	13%	14%	15%
10 000.00	10 000.00	10 000.00	10 000.00	10 000.00	10 000.00
11 000.00	11 100.00	11 200.00	11 300.00	11 400.00	11 500.00
12 100.00	12 321.00	12 544.00	12 769.00	12 996.00	13 225.00
13 310.00	13 676.31	14 049.28	14 428.97	14 815.44	15 208.75
14 641 00	15 180.70	15 735.19	16 304.74	16 889.60	17 490.06
16 105.10	16 850.58	17 623.42	18 424 .35	19 254.15	20 113.57
17 715.61	18 704.15	19 738.23	20 819.52	21 949.73	23 130.61
19 487.17	20 761.60	22 106.81	23 526.05	25 022.69	26 600.20
21 435.89	23 045.38	24 759.63	26 584.44	28 525.86	30 590.23
23 579.48	25 580.37	27 730.79	30 040.42	32 519.49	35 178.76

可以使用一个二维数组（也称为矩阵）存储这些信息。这个数组被命名为balance。

在Java中，声明一个二维数组相当简单。例如：

```
double[][] balances;
```

与一维数组一样，在调用new对多维数组进行初始化之前不能使用它。在这里可以这样初始化：

```
balances = new double[NYEARS][NRATES];
```

另外，如果知道数组元素，就可以不调用new，而直接使用简化的书写形式对多维数组进行初始化。例如：

```
int[][] magicSquare =
   {
      {16, 3, 2, 13},
      {5, 10, 11, 8},
      {9, 6, 7, 12},
      {4, 15, 14, 1}
   };
```

一旦数组被初始化，就可以利用两个方括号访问每个元素，例如，balances[i][j]。

在示例程序中用到了一个存储利率的一维数组interest与一个存储余额的二维数组balances。一维用于表示年，另一维用于表示利率，最初使用初始余额来初始化这个数组的第一行：

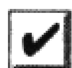
```
for (int j = 0; j < balance[0].length; j++)
   balances[0][j] = 10000;
```

然后，按照下列方式计算其他行：

```
for (int i = 1; i < balances.length; i++)
```

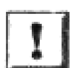
```
{
   for (int j = 0; j < balances[i].length; j++)
   {
      double oldBalance = balances[i - 1][j];
      double interest = . . .;
      balances[i][j] = oldBalance + interest;
   }
}
```

例3-8给出了完整的程序。



注释：for each循环语句不能自动处理二维数组的每一个元素。它是按照行，也就是一维数组处理的。要想访问二维数组 a 的所有元素，需要使用两个嵌套的循环，如下所示：

```
for (double[] row : a)
   for (double value : row)
      do something with value
```



提示：要想快速地打印一个二维数组的数据元素列表，可以调用：

```
System.out.println(Arrays.deepToString(a));
```

输出格式为：

```
[[16, 3, 2, 13], [5, 10, 11, 8], [9, 6, 7, 12], [4, 15, 14, 1]]
```

例3-8 CompoundInterest.java

```
/**
 * This program shows how to store tabular data in a 2D array.
 * @version 1.40 2004-02-10
 * @author Cay Horstmann
 */
public class CompoundInterest
{
   public static void main(String[] args)
   {
      final double STARTRATE = 10;
      final int NRATES = 6;
      final int NYEARS = 10;

      // set interest rates to 10 . . . 15%
      double[] interestRate = new double[NRATES];
      for (int j = 0; j < interestRate.length; j++)
         interestRate[j] = (STARTRATE + j) / 100.0;

      double[][] balances = new double[NYEARS][NRATES];

      // set initial balances to 10000
      for (int j = 0; j < balances[0].length; j++)
         balances[0][j] = 10000;

      // compute interest for future years
      for (int i = 1; i < balances.length; i++)
      {
         for (int j = 0; j < balances[i].length; j++)
         {
            // get last year's balances from previous row
```

```
            double oldBalance = balances[i - 1][j];

            // compute interest
            double interest = oldBalance * interestRate[j];

            // compute this year's balances
            balances[i][j] = oldBalance + interest;
         }
      }

      // print one row of interest rates
      for (int j = 0; j < interestRate.length; j++)
         System.out.printf("%9.0f%%", 100 * interestRate[j]);

      System.out.println();

      // print balance table
      for (double[] row : balances)
      {
         // print table row
         for (double b : row)
            System.out.printf("%10.2f", b);

         System.out.println();
      }
   }
}
```

3.10.7 不规则数组

到目前为止，读者所看到的数组与其他程序设计语言中提供的数组没有多大区别。但实际存在着一些细微的差异，而这正是Java的优势所在：Java实际上没有多维数组，只有一维数组。多维数组被解释为“数组的数组。”

例如，在前面的示例中，balances数组实际上是一个包含10个元素的数组，而每个元素又是一个由6个浮点数组成的数组（请参看图3-16）。

表达式balances[i]引用第i个子数组，也就是二维表的第i 行。它本身也是一个数组，balances[i][j]引用这个数组的第j项。

由于可以单独地存取数组的某一行，所以可以让两行交换。

```
double[] temp = balances[i];
balances[i] = balances[i + 1];
balances[i + 1] = temp;
```

还可以方便地构造一个“不规则”数组，即数组的每一行有不同的长度。下面是一个典型的示例。在这个示例中，创建一个数组，第i行第j列将存放“从i个数值中抽取j个数值”产生的结果。

```
1
1  1
1  2  1
1  3  3  1
1  4  6  4  1
1  5  10 10 5  1
1  6  15 20 15 6  1
```

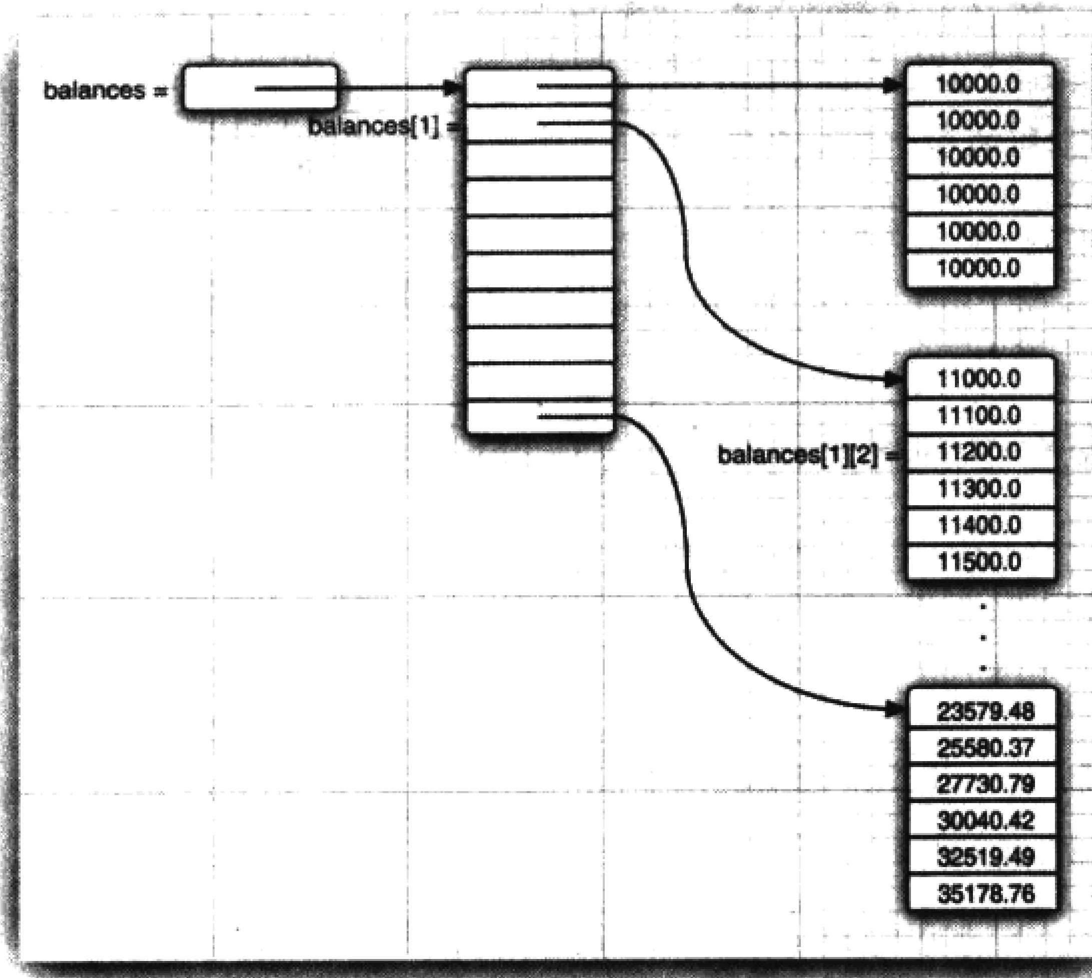


图3-16 一个二维数组

由于j不可能大于i，所以矩阵是三角形的。第i行有i + 1个元素（允许抽取0个元素，也是一种选择）。要想创建一个不规则的数组，首先需要分配一个具有所含行数的数组。

```
int[][] odds = new int[NMAX + 1][];
```

接下来，分配这些行。

```
for (int n = 0; n <= NMAX; n++)
   odds[n] = new int[n + 1];
```

在分配了数组之后，假定没有超出边界，就可以采用通常的方式访问其中的元素了。

```
for (int n = 0; n < odds.length; n++)
   for (int k = 0; k < odds[n].length; k++)
   {
      // compute lotteryOdds
      . . .
      odds[n][k] = lotteryOdds;
   }
```

例3-9给出了完整的程序。


C++注释：在C++中，Java的声明

```
double[][] balances = new double[10][6]; // Java
```

不同于

```
double balances[10][6]; // C++
```

也不同于

```
double (*balances)[6] = new double[10][6]; // C++
```

而是创建了一个包含10个指针的一个数组：

```
double** balances = new double*[10]; // C++
```

然后，指针数组的每一个元素被分配了一个包含6个数值的数组：

```
for (i = 0; i < 10; i++)
   balances[i] = new double[6];
```

庆幸的是，当创建new double[10][6]时，这个循环将自动地执行。当需要不规则的数组时，只能单独地创建行数组。

例3-9 LotteryArray.java

```
/**
 * This program demonstrates a triangular array.
 * @version 1.20 2004-02-10
 * @author Cay Horstmann
 */
public class LotteryArray
{
   public static void main(String[] args)
   {
      final int NMAX = 10;

      // allocate triangular array
      int[][] odds = new int[NMAX + 1][];
      for (int n = 0; n <= NMAX; n++)
         odds[n] = new int[n + 1];

      // fill triangular array
      for (int n = 0; n < odds.length; n++)
         for (int k = 0; k < odds[n].length; k++)
         {
            /*
             * compute binomial coefficient n*(n-1)*(n-2)*...*(n-k+1)/(1*2*3*...*k)
             */
            int lotteryOdds = 1;
            for (int i = 1; i <= k; i++)
               lotteryOdds = lotteryOdds * (n - i + 1) / i;

            odds[n][k] = lotteryOdds;
         }

      // print triangular array
      for (int[] row : odds)
      {
         for (int odd : row)
            System.out.printf("%4d", odd);
         System.out.println();
      }
   }
}
```

现在，已经看到了Java语言的基本程序结构，下一章节将介绍Java中的面向对象的程序设计。

第4章 对象与类

- ▲ 面向对象程序设计概述
- ▲ 使用现有类
- ▲ 用户自定义类
- ▲ 静态域和方法
- ▲ 方法参数
- ▲ 对象构造
- ▲ 包
- ▲ 类路径
- ▲ 文档注释
- ▲ 类设计技巧

这一章将主要介绍如下内容：

- 面向对象程序设计
- 如何创建标准Java类库中的类对象
- 如何编写自己的类

如果你没有面向对象程序设计的应用背景，就一定要认真地阅读一下本章的内容。面向对象程序设计与面向过程程序设计在思维方式上存在着很大的差别。改变一种思维方式并不是一件很容易的事情，而且为了继续学习Java也要清楚对象的概念。

对于具有使用C++经历的程序员来说，与上一章相同，对本章的内容不会感到太陌生，但在两种语言之间还是存在着很多不同之处，所以要认真地阅读本章的后半部分内容，你将发现"C++注释"对学习Java语言会很有帮助的。

4.1 面向对象程序设计概述

面向对象程序设计（简称OOP）是当今主流的程序设计范型，它已经取代了70年代的"结构化"过程化程序设计开发技术。Java 是完全面向对象的，必须熟悉OOP才能够编写Java程序。

面向对象的程序是由对象组成的，每个对象包含对用户公开的特定功能部分和隐藏的实现部分。程序中的很多对象来自于标准库，还有一些是自定义的。究竟是自己构造对象，还是从外界购买对象完全取决于预算和时间。但是，从根本上说，只要对象能够满足要求，就不必关心其功能的具体实现过程。在OOP中，不必关心对象的具体实现，只要能够满足用户的需求即可。

传统的结构化程序设计通过设计一系列的过程（即算法）来求解问题。这些过程一旦被确定，就要开始考虑存储数据的方式。这就是Pascal语言的设计者Niklaus Wirth将其编著的有关程序设计的著名书籍命名为《算法＋数据结构＝程序》（*Algorithms + Data Structures = Programs*, Prentice Hall, 1975）的原因。需要注意的是，在Wirth命名的标题中，算法是第一位的，数据结构是第二位的。这就明确地表述了程序员的工作方式。首先要确定如何操作数据，然后再决定如何组织数据，以便于数据操作。OOP却调换了这个次序，数据被放在第一位，然后再考虑操作数据的算法。

对于一些规模较小的问题，将其分解为过程的开发方式比较理想。而面向对象更加适用于解决规模较大的问题。要想实现一个简单的web浏览器可能需要大约2000个过程，这些过程可能需要对一组全局数据进行操作。采用面向对象的设计风格，可能只需要大约100个类，每个类平均包含20个方法（如图4-1所示）。后者更易于程序员掌握，也容易找到bug。假设给定对象的数据处于一种错误状态，在访问过这个数据项的20个方法中查找错误要比在2000个过程中查找容易得多。

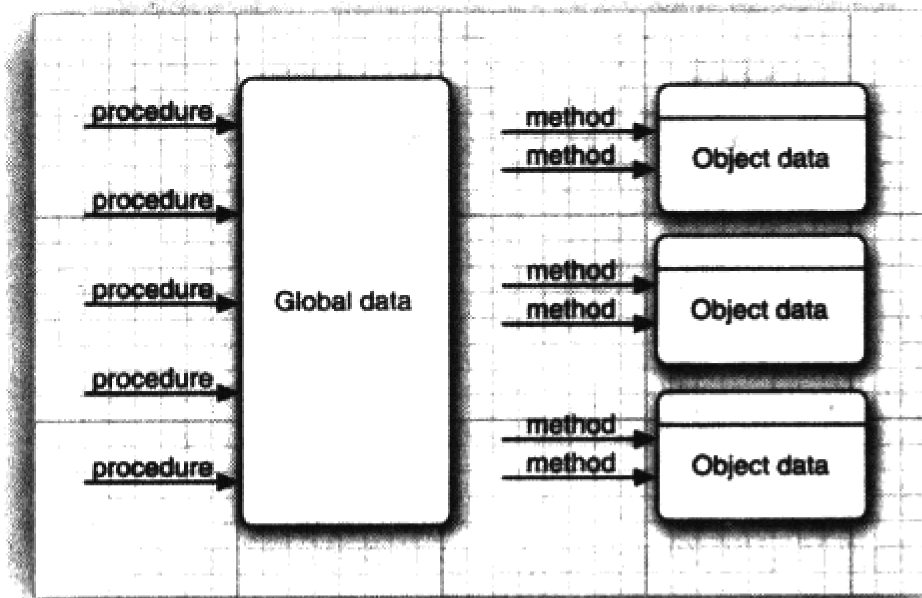


图4-1 面向过程与面向对象的程序设计对比

4.1.1 类

类（class）是构造对象的模板或蓝图。我们可以将类想像成小甜饼的切割机，将对象想像为小甜饼。由类*构造*（construst）对象的过程称为创建类的*实例*（instance）。

正如前面所看到的，用Java编写的所有代码都位于某个类的内部。标准的Java库提供了几千个类，可以用于用户界面设计、日期、日历和网络程序设计。尽管如此，还是需要在Java程序中创建一些自己的类，以便描述应用程序所对应的问题域中的对象。

封装（encapsulation，有时称为数据隐藏）是与对象有关的一个重要概念。从形式上看，封装不过是将数据和行为组合在一个包中，并对对象的使用者隐藏了数据的实现方式。对象中的数据称为*实例域*（instance fields），操纵数据的过程称为*方法*（method）。对于每个特定的类实例（对象）都有一组特定的实例域值。这些值的集合就是这个对象的当前*状态*（state）。无论何时，只要向对象发送一个消息，它的状态就有可能发生改变。

实现封装的关键在于*绝对不能*让类中的方法直接地访问其他类的实例域。程序仅通过对象的方法与对象数据进行交互。封装给予对象了“黑盒”特征，这是提高重用性和可靠性的关键。这意味着一个类可以全面地改变存储数据的方式，只要仍旧使用同样的方法操作数据，其他对象就不会知道或介意所发生的变化。

OOP的另一个原则会让用户自定义Java类变得轻而易举，这就是：类可以通过*扩展*另一个类来建立。事实上，在Java中，所有的类都源自于一个“神通广大的超类”，它就是 Object。在下一章中，读者将可以看到有关Object类的详细介绍。

在对一个已有的类扩展时，这个扩展后的新类具有所扩展的类的全部属性和方法。在新类中，只需要提供那些仅适用于这个类的新方法和数据域就可以了。通过扩展一个类来建立另外一个类的过程被称为继承（inheritance），有关继承的详细内容请参看下一章。

4.1.2 对象

要想使用OOP，一定要清楚对象的三个主要特性：

- 对象的行为（behavior）——可以对对象施加哪些操作，或可以对对象施加哪些方法？
- 对象的状态（state）——当施加那些方法时，对象如何响应？
- 对象标识（identity）——如何辨别具有相同行为与状态的不同对象？

同一个类的所有对象实例，由于支持相同的行为而具有家族相似性。对象的行为是用可调用的方法定义的。

此外，每个对象都保存着描述当前特征的信息。这就是对象的状态。对象的状态可能会随着时间而发生改变，但这种改变不会是自发的。对象状态的改变必须通过调用方法实现（如果不经过方法调用就可以改变对象状态，只能说明封装性遭到了破坏）。

但是，对象的状态并不能完全描述一个对象。每个对象都有一个惟一的身份（identity）。例如，在一个订单处理系统中，任何两个订单都存在着不同之处，即使所订购的货物完全相同也是如此。需要注意，作为一个类的实例，每个对象的标识永远是不同的，状态常常也存在着差异。

对象的这些关键特性在彼此之间相互影响着。例如，对象的状态影响它的行为（如果一个订单“已送货”或“已付款”，就应该拒绝调用具有增删订单中条目的方法。反过来，如果订单是“空的”，即还没有加入预订的物品，这个订单就不应该进入“已送货”状态）。

4.1.3 识别类

传统的过程化程序设计，必须从顶部的main函数开始编写程序。在设计面向对象的系统时没有所谓的“顶部”。对于学习OOP的初学者来说常常会感觉无从下手。答案是：首先从设计类开始，然后再往每个类中添加方法。

识别类的简单规则是在分析问题的过程中寻找名词，而方法对应着动词。

例如，在订单处理系统中，有这样一些名词：

- 项目（Item）
- 订单（Order）
- 送货地址（Shipping address）
- 付款（Payment）
- 账户（Account）

这些名词很可能成为类Item、Order等等。

接下来，查看动词。物品项目被添加到订单中。订单被发送或取消。订单货款被支付。对于每一个动词如：“添加”、“发送”、“取消”以及“支付”，都要标识出主要负责完成相应动作的对象。例如，当一个新的条目添加到订单中时，那个订单对象就是被指定的对象，因为它知道如何存储条目以及如何对条目进行排序。也就是说，add应该是Order类的一个方法，而Item

对象是一个参数。

当然，所谓“名词与动词”原则只是一种粗略的方法，在创建类的时候，哪些名词和动词是重要的完全取决于个人的开发经验。

4.1.4 类之间的关系

在类之间，最常见的关系有

- 依赖（“uses-a”）
- 聚合（“has-a”）
- 继承（“is-a”）

依赖（dopendence），即“uses-a”关系，是一种最明显的、最常见的关系。例如，Order类使用Account类是因为Order对象需要访问Account对象查看信用状态。但是Item类不依赖于Account类，这是因为Item对象与客户账户无关。因此，如果一个类的方法操纵另一个类的对象，我们就说一个类依赖于另一个类。

应该尽可能地将相互依赖的类减至最少。如果类A不知道B的存在，它就不会关心B的任何改变（这意味着B的改变不会导致A产生任何bug）。用软件工程的术语来说，就是让类之间的耦合度最小。

聚合（aggregation），即“has-a”关系，是一种具体且易于理解的关系。例如，一个Order对象包含一些Item对象。聚合关系意味着类A的对象包含类B的对象。

注释：有些方法学家不喜欢聚合这个概念，而更加喜欢使用“关联”。从建模的角度看，这是可以理解的。但对于程序员来说，“has-a”显得更加形象。喜欢使用聚合的另一个理由是关联的标准符号不易区分。请参看表4-1。

表4-1 表达类关系的UML符号

关系	UML连接符
继承	———————▷
接口继承	- - - - - - - -▷
依赖	- - - - - - - ->
聚合	◇———————
关联	————————
直接关联	———————>

继承（inheritance），即“is-a”关系，是一种用于表示特殊与一般关系的。例如，RushOrder类由Order类继承而来。在具有特殊性的RushOrder类中包含了一些用于优先处理的特殊方法，以及一个计算运费的不同方法；而其他的方法，如添加条目、生成账单等等都是从Order类继承来的。一般而言，如果类A扩展类B，类A不但包含从类B继承的方法，还会拥有一些额外的功能（下一章将详细讨论继承，其中会用较多的篇幅讲述这个重要的概念）。

很多程序员采用UML（Unified Modeling Language）绘制描述类之间关系的类图。图4-2就

是这样一个例子。类用矩形表示，类之间的关系用带有各种修饰的箭头表示。表4-1给出了UML中最常见的箭头样式。

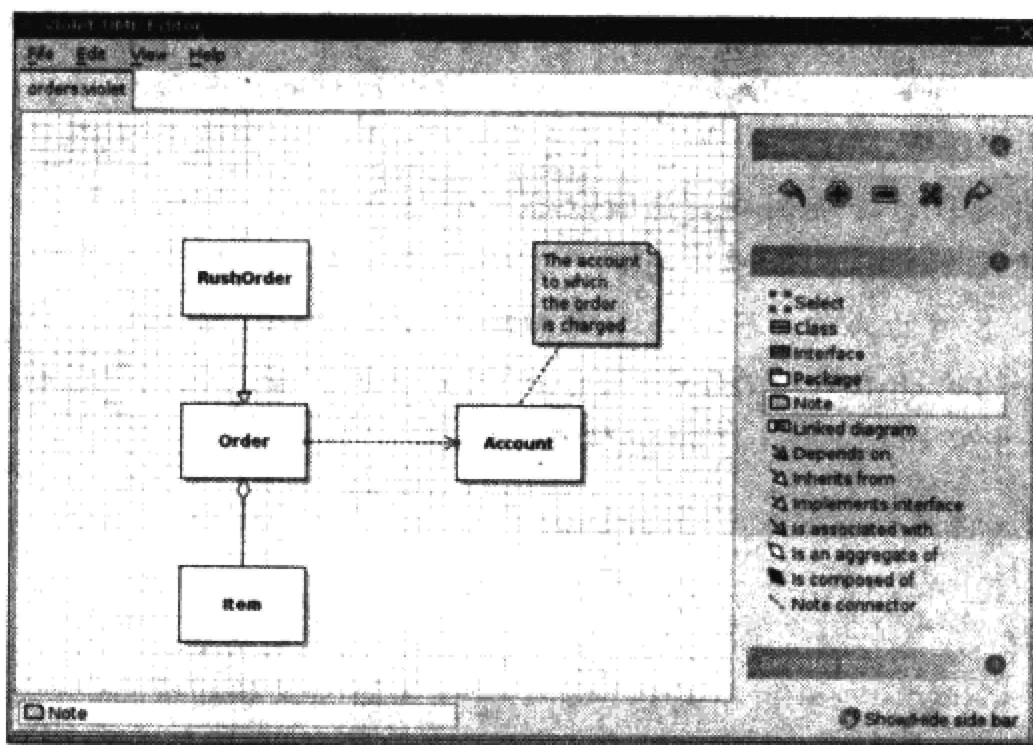


图4-2 类的示意图

注释：有很多专门用来绘制UML类图的工具。有些开发商提供了一些高性能（当然也是昂贵的）的工具旨在辅助开发过程。其中，有Rational Rose（http://www.ibm.com/software/awdtools/developer/Rose）和Together（http://www.borland.com/us/products/together）。还有一种选择是使用开放源代码程序ArgoUML（http://argouml.tigris.org）。在GentleWare（http://gentleware.com）提供了一个商业支持版本。如果只想使用UML最粗浅的内容绘制一个简单的UML类图，就可以试试Violet（http://violet.sourceforge.net）。

4.2 使用现有类

在Java中，没有类就无法做任何事情，我们前面曾经接触过几个类。然而，并不是所有的类都具有面向对象特征。例如，Math类。在程序中，可以使用Math类的方法，如 Math.random，并只需要知道方法名和参数（如果有的话），而不必了解它的具体实现过程。这正是封装的关键所在，当然所有类都是这样。但遗憾的是，Math 类只封装了功能，它不需要也不必隐藏数据。由于没有数据，因此也不必担心生成对象以及初始化实例域。

下一节将会给出一个更加具有代表性的类——Date类。从中可以看到如何构造对象，以及如何调用类的方法。

4.2.1 对象与对象变量

要想使用对象，就必须首先构造对象，并指定其初始状态。然后，对对象施加方法。

在Java程序设计语言中，使用*构造器*（constructor）构造新实例。构造器是一种特殊的方法，用来构造并初始化对象。下面看一个例子。在标准Java库中包含一个Date类。它的对象将描述一个时间点，例如："December 31, 1999, 23:59:59 GMT"。

注释：可能会感到奇怪：为什么用类描述时间，而不像其他语言那样用一个内置的（built-in）类型？例如，在Visual Basic中有一个内置的date类型，程序员可以采用#6/1/1995#格式指定日期。从表面上看，这似乎很方便，因为程序员使用内置的date类型，而不必为设计类而操心。但实际上，Visual Basic这样设计的适应性如何呢？在有些地区日期表示为月/日/年，而另一些地区表示为日/月/年。语言设计者是否能够预见这些问题呢？如果没有处理好这类问题，语言就有可能陷入混乱，对此感到不满的程序员也会丧失使用这种语言的热情。如果使用类，这些设计任务就交给了类库的设计者。如果类设计的不完善，其他的操作员可以很容易地编写自己的类，以便增强或替代（replace）系统提供的类（作为这个问题的印证：Java的日期类库有些混乱，对它的重新设计正在进行中。请参看http://jcp.org/en/jsr/detail?id=310）。

构造器的名字应该与类名相同。因此Date类的构造器名为Date。要想构造一个Date对象，需要在构造器前面加上new操作符，如下所示：

```
new Date()
```

这个表达式构造了一个新对象。这个对象被初始化为当前的日期和时间。

如果需要的话，也可以将这个对象传递给一个方法：

```
System.out.println(new Date());
```

相反，也可以将一个方法应用于刚刚创建的对象上。Date类中有一个toString方法。这个方法将返回日期的字符串描述。下面的语句可以说明如何将toString方法应用于新构造的 Date 对象上。

```
String s = new Date().toString();
```

在这两个例子中，构造的对象仅使用了一次。通常，希望构造的对象可以多次使用，因此，需要将对象存放在一个变量中：

```
Date birthday = new Date();
```

图4-3显示了引用新构造的对象变量birthday。

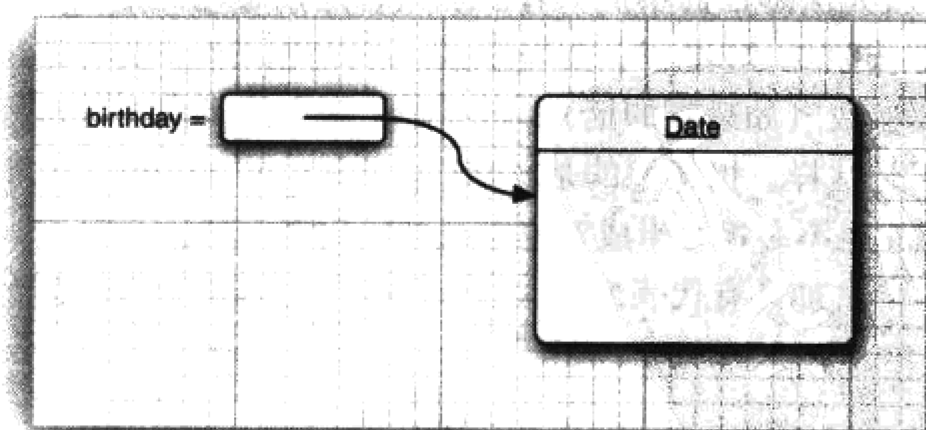


图4-3 创建一个新对象

在对象与对象变量之间存在着一个重要的区别。例如，语句

```
Date deadline; // deadline doesn't refer to any object
```

定义了一个对象变量deadline，它可以引用Date类型的对象。但是，一定要认识到：变量deadline

不是一个对象，实际上也没有引用对象。此时，不能将任何 Date 方法应用于这个变量上。语句

```
s = deadline.toString(); // not yet
```

将产生编译错误。

必须首先初始化变量deadline。这里有两个选择。当然，可以用新构造的对象初始化这个变量：

```
deadline = new Date();
```

也让这个变量引用一个已存在的对象：

```
deadline = birthday;
```

现在，这两个变量引用同一个对象（请参见图4-4）。

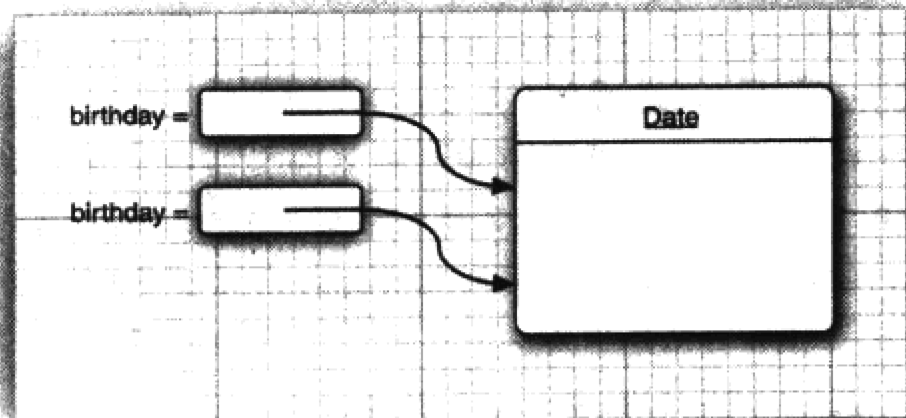


图4-4 引用同一个对象的对象变量

一定要认识到：一个对象变量并没有实际包含一个对象，而仅仅引用一个对象。

在Java中，任何对象变量的值都是对存储在另外一个地方的一个对象的引用。new操作符的返回值也是一个引用。下列语句：

```
Date deadline = new Date();
```

有两个部分。表达式new Date()构造了一个Date类型的对象，并且它的值是对新创建对象的引用。这个引用存储在变量deadline中。

可以显式地将对象变量设置为null，表明这个对象变量目前没有引用任何对象。

```
deadline = null;
. . .
if (deadline != null)
   System.out.println(deadline);
```

如果将一个方法应用于一个值为null的对象上，那么就会产生运行错误。

```
birthday = null;
String s = birthday.toString(); // runtime error!
```

变量不会自动地初始化为null，而必须通过调用new 或将它们设置为null进行初始化。

C++注释：很多人错误地认为Java对象变量与C++的引用类似。然而，在C++ 中没有空引用，并且引用不能被赋值。可以将Java的对象变量看作C++的对象指针。例如，

```
Date birthday; // Java
```

实际上，等同于

```
Date* birthday; // C++
```

一旦理解了这一点，一切问题就迎刃而解了。当然，一个Date*指针只能通过调用new进行初始化。就这一点而言，C++与Java的语法几乎是一样的。

```
Date* birthday = new Date(); // C++
```

如果把一个变量的值赋给另一个变量，两个变量就指向同一个日期，即指向同一个对象。在Java中的null引用对应C++中的NULL指针。

所有的Java对象都存储在堆中。当一个对象包含另一个对象变量时，这个变量依然包含着指向另一个堆对象的指针。

在C++中，指针十分令人头疼，并常常导致程序错误。稍不小心就会创建一个错误的指针，或者造成内存溢出。在Java语言中，这些问题都不复存在。如果使用一个没有初始化的指针，运行系统将会产生一个运行时错误，而不是生成一个随机的结果。同时，不必担心内存管理问题，垃圾收集器将会处理相关的事宜。

C++确实做了很大的努力，它通过拷贝型构造器和复制操作符来实现对象的自动拷贝。例如,一个链表（linked list）拷贝的结果将会得到一个新链表，其内容与原始链表相同，但却是一组独立的链接。这使得将同样的拷贝行为内置在类中成为可能。在Java中，必须使用clone方法获得对象的完整拷贝。

4.2.2 Java类库中的GregorianCalendar类

在前面的例子中，已经使用了Java标准类库中的Date类。Date类的实例有一个状态，即特定的时间点。

尽管在使用Date类时不必知道这一点，但时间是用距离一个固定时间点的毫秒数（可正可负）表示的，这个点就是所谓的纪元（epoch），它是UTC时间1970年1月1日 00:00:00。UTC是Coordinated Universal Time的缩写，与大家熟悉的GMT （即 Greenwich Mean Time/格林威治时间）一样，是一种具有实际目的的科学标准时间。

但是，Date 类所提供的日期处理并没有太大的用途。Java类库的设计者认为：像“December 31, 1999, 23:59:59”这样的日期表示法只是阳历的固有习惯。这种特定的描述法遵循了世界上大多数地区使用的Gregorian阳历表示法。但是，同一时间点采用中国的农历表示和采用希伯来的阴历表示就太不一样，对于火星历来说就更不可想像了。



注释：有史以来，人类的文明与历法的设计紧紧地相连，日历给日期命名、给太阳和月亮的周期排列次序。有关世界上各种日历的有趣解释，从法国革命的日历到玛雅人计算日期的方法等等，请参看Nachum Dershowitz 和 Edward M. Reingold 编写的《Calendrical Calculations》（剑桥大学出版社，第2版，2001年）。

类库设计者决定将保存时间与给时间点命名分开。所以标准Java类库分别包含了两个类：一个是用来表示时间点的Date类；另一个是用来表示大家熟悉的日历表示法的GregorianCalendar类。事实上，GregorianCalendar类扩展了一个更加通用的Calendar类，这个类描述了日历的一般属性。理论上，可以通过扩展Calendar类来实现中国的阴历或者是火星日历。然而，标准类库中只实现了Gregorian日历。

将时间与日历分开是一种很好的面向对象设计。通常，最好使用不同的类表示不同的概念。

Date类只提供了少量的方法用来比较两个时间点。例如 before 和 after 方法分别表示一个时间点是否早于另一个时间点，或者晚于另一个时间点。

```
if (today.before(birthday))
   System.out.println("Still time to shop for a gift.");
```

注释：实际上，Date类还有getDay、getMonth以及getYear等方法，然而并不推荐使用这些方法。当类库设计者意识到某个方法不应该存在时，就把它标记为不鼓励使用。

当类库的设计者意识到单独设计日历类更有实际意义时，这些方法已经是Date类的一部分。引入日历类之后，Date类中的这些方法被标明为不鼓励使用，虽然在程序中仍然可以使用它们，但是如果这样做，编译时会出现警告。最好还是不要使用这部分方法，它们有可能会从未来的类库版本中删去。

GregorianCalendar类所包含的方法要比Date类多得多。特别是有几个很有用的构造器。表达式

```
new GregorianCalendar()
```

构造一个新对象，用于表示对象构造时的日期和时间。

另外，还可以通过提供年、月、日构造一个表示某个特定日期午夜的日历对象：

```
new GregorianCalendar(1999, 11, 31)
```

有些怪异的是，月份从0 开始计数。因此11 表示十二月。为了清晰起见，也可以使用常量，如：Calendar.DECEMBER。

```
new GregorianCalendar(1999, Calendar.DECEMBER, 31)
```

还可以设置时间：

```
new GregorianCalendar(1999, Calendar.DECEMBER, 31, 23, 59, 59)
```

当然，常常希望将构造的对象存储在对象变量中：

```
GregorianCalendar deadline = new GregorianCalendar(. . .);
```

GregorianCalendar类封装了实例域，这些实例域保存着设置的日期信息。不查看源代码不可能知道日期在类中的具体表达方式。当然，这一点并不重要，重要的是要知道类向外界开放的方法。

4.2.3 更改器方法与访问器方法

现在，可能有人会问：如何从封装在某个GregorianCalendar 对象内部的日期中获得当前的日、月、年呢？如果希望对这些内容做一些修改，又该怎么做呢？在联机文档中，以及本章末尾的API注释中可以找到答案。这里只讲述一些最重要的方法。

日历的作用是提供某个时间点的年、月、日等信息。要想查询这些设置信息，应该使用GregorianCalendar类的get方法。为了表达希望得到的项，需要借助于Calendar类中定义的一些常量，如：Calendar.MONTH或Calendar.DAY_OF_WEEK：

```
GregorianCalendar now = new GregorianCalendar();
int month = now.get(Calendar.MONTH);
```

```
int weekday = now.get(Calendar.DAY_OF_WEEK);
```

API注释列出了可以使用的全部常量。

调用set方法，可以改变对象的状态：

```
deadline.set(Calendar.YEAR, 2001);
deadline.set(Calendar.MONTH, Calendar.APRIL);
deadline.set(Calendar.DAY_OF_MONTH, 15);
```

另外，还有一个便于设置年、月、日的方法，调用方式如下：

```
deadline.set(2001, Calendar.APRIL, 15);
```

最后，还可以为给定的日期对象增加天数、星期数、月份等等：

```
deadline.add(Calendar.MONTH, 3); // move deadline by 3 months
```

如果传递的数值是一个负数，日期就向后移。

get方法与set和add方法在概念上是有区别的。get方法仅仅查看并返回对象的状态，而set和add方法却对对象的状态进行修改。对实例域做出修改的方法被称为*更改器方法*（mutator method），仅访问实例域而不进行修改的方法被称为*访问器方法*（accessor method）。

C++注释：*在C++中，带有const 后缀的方法是访问器方法；默认为更改器方法。但是，在Java语言中，访问器方法与更改器方法在语法上没有明显的区别。*

通常的习惯是在访问器方法名前面加上前缀 get，在更改器方法前面加上前缀 set。例如，在GregorianCalendar类有 getTime 方法和 setTime方法，它们分别用来获得和设置日历对象所表示的时间点。

```
Date time = calendar.getTime();
calendar.setTime(time);
```

这些方法在进行GregorianCalendar和Date类之间的转换时非常有用。这里有一个例子。假定已知年、月、日，并且希望创建一个包含这个时间值的Date对象。由于Date类不知道如何操作日历，所以首先需要构造一个GregorianCalendar对象，然后再调用getTime方法获得一个日期对象：

```
GregorianCalendar calendar = new GregorianCalendar(year, month, day);
Date hireDay = calendar.getTime();
```

与之相反，如果希望获得Date对象中的年、月、日信息，就需要构造一个GregorianCalendar对象、设置时间，然后再调用get方法：

```
GregorianCalendar calendar = new GregorianCalendar();
calendar.setTime(hireDay);
int year = calendar.get(Calendar.YEAR);
```

下面用一个应用GregorianCalendar类的程序来结束本节内容的论述。这个程序将显示当前月的日历，其格式为：

```
Sun Mon Tue Wed Thu Fri Sat
                          1
  2   3   4   5   6   7   8
  9  10  11  12  13  14  15
 16  17  18  19* 20  21  22
 23  24  25  26  27  28  29
 30  31
```

当前的日用一个*号标记。可以看到，这个程序需要解决如何计算某月份的天数以及一个给定日期相应是星期几。

下面看一下这个程序的关键步骤。首先，构造了一个日历对象，并用当前的日期和时间进行初始化。

```
GregorianCalendar d = new GregorianCalendar();
```

经过两次调用get方法获得当时的日、月。

```
int today = d.get(Calendar.DAY_OF_MONTH);
int month = d.get(Calendar.MONTH);
```

然后，将d设置为这个月的第一天，并得到这一天为星期几。

```
d.set(Calendar.DAY_OF_MONTH, 1);
int weekday = d.get(Calendar.DAY_OF_WEEK);
```

如果这个月的第一天是星期日，变量weekday就设置为Calendar.SUNDAY；如果这个月的第一天是星期一，就设置为Calendar.MONDAY，以此类推（实际上，这些值是整数值1，2，…，7，但最好不要依赖背景知识书写代码）。

注意，在日历的第一行是缩进的，因此，月份的第一天指向相应的星期几。由于每个星期的第一天存在着不同的约定习惯，因此需要能够随机应变。在美国，每个星期的第一天是星期日；在欧洲，每个星期的第一天是星期一，最后一天是星期日。

Java虚拟机非常注意当前用户的所在地区，不同的地区存在着不同的格式习惯，包括每星期的起始日以及一星期中每天的命名方式。

提示：如果想看到不同地区程序的输出，应该在main方法的第一行中添加下列代码：

```
Locale.setDefault(Locale.ITALY);
```

getFirstDayOfWeek方法获得当前地区星期的起始日。为了确定所需要的缩进距离，将日历对象的日减1，直到一个星期的第一天为止。

```
int firstDayOfWeek = d.getFirstDayOfWeek();
int indent = 0;
while (weekday != firstDayOfWeek)
{
   indent++;
   d.add(Calendar.DAY_OF_MONTH, -1);
   weekday = d.get(Calendar.DAY_OF_WEEK);
}
```

随后，输出表示星期几名称的头。这个操作可以通过调用DateFormatSymbols类方法实现。

```
String [] weekdayNames = new DateFormatSymbols().getShortWeekdays();
```

getShortWeekdays方法返回用户语种所命名的表示星期几的缩写字符串（例如：英语将返回“Sun”、“Mon”等）。数组用星期数作为索引。下面的循环将打印标题：

```
do
{
   System.out.printf("%4s", weekdayNames[weekday]);
   d.add(Calendar.DAY_OF_MONTH, 1);
   weekday = d.get(Calendar.DAY_OF_WEEK);
}
```

```
while (weekday != firstDayOfWeek);
System.out.println();
```

现在，已经做好打印日历内容的准备了。第一行缩进，并将日期对象设置为月份的起始日。在循环中用d记录一个月中的每一天。

在每次迭代过程中，打印日期值。如果d是当前日期，打印日期之后再打印一个*标记。每个星期的第一天，重新换行打印。而后，将d设置为下一天：

```
d.add(Calendar.DAY_OF_MONTH, 1);
```

什么时候结束呢？我们并不知道这个月有31天、30天、29天，还是28天。实际上，只要d还指示当月就应该继续迭代。

```
do
{
   . . .
}
while (d.get(Calendar.MONTH) == month);
```

一旦d进入下一个月，程序就终止执行。

例4-1给出了完整的程序。

正如前面所看到的，日历程序包含了一些复杂问题，例如：某一天是星期几，每个月有多少天等等。有了 GregorianCalendar 类一切就变得简单了。我们并不必知道 GregorianCalendar 类是如何计算星期数和每个月的天数，而只需要使用类提供的接口：get方法、set方法以及 add 方法就可以了。

这个示例程序关键告诉我们：可以使用类的接口解决复杂任务，而不必知道其中的实现细节。

例4-1 CalendarTest.java

```
import java.text.DateFormatSymbols;
import java.util.*;

/**
 * @version 1.4 2007-04-07
 * @author Cay Horstmann
 */

public class CalendarTest
{
   public static void main(String[] args)
   {
      // construct d as current date
      GregorianCalendar d = new GregorianCalendar();

      int today = d.get(Calendar.DAY_OF_MONTH);
      int month = d.get(Calendar.MONTH);

      // set d to start date of the month
      d.set(Calendar.DAY_OF_MONTH, 1);

      int weekday = d.get(Calendar.DAY_OF_WEEK);

      // get first day of week (Sunday in the U.S.)
      int firstDayOfWeek = d.getFirstDayOfWeek();
```

```java

      // determine the required indentation for the first line
      int indent = 0;
      while (weekday != firstDayOfWeek)
      {
         indent++;
         d.add(Calendar.DAY_OF_MONTH, -1);
         weekday = d.get(Calendar.DAY_OF_WEEK);
      }

      // print weekday names
      String[] weekdayNames = new DateFormatSymbols().getShortWeekdays();
      do
      {
         System.out.printf("%4s", weekdayNames[weekday]);
         d.add(Calendar.DAY_OF_MONTH, 1);
         weekday = d.get(Calendar.DAY_OF_WEEK);
      }
      while (weekday != firstDayOfWeek);
      System.out.println();

      for (int i = 1; i <= indent; i++)
         System.out.print("    ");

      d.set(Calendar.DAY_OF_MONTH, 1);
      do
      {
         // print day
         int day = d.get(Calendar.DAY_OF_MONTH);
         System.out.printf("%3d", day);

         // mark current day with *
         if (day == today) System.out.print("*");
         else System.out.print(" ");

         // advance d to the next day
         d.add(Calendar.DAY_OF_MONTH, 1);
         weekday = d.get(Calendar.DAY_OF_WEEK);

         // start a new line at the start of the week
         if (weekday == firstDayOfWeek) System.out.println();
      }
      while (d.get(Calendar.MONTH) == month);
      // the loop exits when d is day 1 of the next month

      // print final end of line if necessary
      if (weekday != firstDayOfWeek) System.out.println();
   }
}
```

API java.util.GregorianCalendar 1.1

- GregorianCalendar()

  构造一个日历对象，用来表示默认地区、默认时区的当前时间。

- GregorianCalendar(int year, int month, int day)

- `GregorianCalendar(int year, int month, int day, int hour, int minutes, int seconds)`

  用给定的日期和时间构造一个Gregorian 日历对象。

  参数：year　　该日期所在的年份
  　　　month　　该日期所在的月份。此值以0为基准；例如，0表示一月
  　　　day　　该月份中的日期
  　　　hour　　小时（0到23之间）
  　　　minutes　　分钟（0到59之间）
  　　　seconds　　秒（0到59之间）

- `int get(int field)`

  返回给定域的值。

  参数：field　　可以是下述选项之一：Calendar.ERA、Calendar.YEAR、Calendar.MONTH、Calendar.WEEK_OF_YEAR、Calendar.WEEK_OF_MONTH、Calendar.DAY_OF_MONTH、Calendar.DAY_OF_YEAR、Calendar.DAY_OF_WEEK、Calendar.DAY_OF_WEEK_IN_MONTH、Calendar.AM_PM、Calendar.HOUR、Calendar.HOUR_OF_DAY、Calendar.MINUTE、Calendar.SECOND、Calendar.MILLISECOND、Calendar.ZONE_OFFSET、Calendar.DST_OFFSET

- `void set(int field, int value)`

  设置特定域的值。

  参数：field　　get接收的常量之一
  　　　value　　新值

- `void set(int year, int month, int day)`
- `void set(int year, int month, int day, int hour, int minutes, int seconds)`

  将日期域和时间域设置为新值。

  参数：year　　该日期所在的年份
  　　　month　　该日期所在的月份。此值以0为基准；例如，0表示一月
  　　　day　　该月份中的日期
  　　　hour　　小时（0到23）
  　　　minutes　　分钟（0到59）
  　　　seconds　　秒（0到59）

- `void add(int field, int amount)`

  这是一个可以对日期信息实施算术运算的方法。对给定的时间域增加指定数量的时间。例如，可以通过调用c.add(Calendar.DAY_OF_MONTH, 7)，将当前的日历日期加上7。

  参数：field　　需要修改的域（可以使用get方法文档中给出的一个常量）
  　　　amount　　域被改变的数量（可以是负值）

- `int getFirstDayOfWeek()`

  获得当前用户所在地区，一个星期中的第一天。例如：在美国一个星期中的第一天是Calendar.SUNDAY。
- `void setTime(Date time)`

  将日历设置为指定的时间点。

  参数：time 时间点
- `Date getTime()`

  获得这个日历对象当前值所表达的时间点。

API java.text.DateFormatSymbols 1.1

- `String[] getShortWeekdays()`
- `String[] getShortMonths()`
- `String[] getWeekdays()`
- `String[] getMonths()`

获得当前地区的星期几或月份的名称。利用Calendar的星期和月份常量作为数组索引值。

4.3 用户自定义类

在第3章中，已经开始编写了一些简单的类。但是，那些类都只包含一个简单的main方法。现在开始学习如何设计复杂应用程序所需要的各种主力类（workhorse class）。通常，这些类没有main方法，而却有自定义的实例域和实例方法。要想创建一个完整的程序，应该将若干类组合在一起，其中只有一个类有main方法。

4.3.1 一个Employee类

在Java中，最简单的类定义形式为：

```
class ClassName
{
   constructor1
   constructor2
   . . .
   method1
   method2
   . . .
   field1
   field2
   . . .
}
```



注释：这里编写类所采用的风格是类的方法在前面，域在后面。这种风格有易于促使人们更加关注接口的概念，削减对实现的注意。

下面看一个非常简单的Employee类。在编写薪金管理系统时可能会用到。

```
class Employee
{
```

```
    // constructor
    public Employee(String n, double s, int year, int month, int day)
    {
       name = n;
       salary = s;
       GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
       hireDay = calendar.getTime();
    }

    // a method
    public String getName()
    {
       return name;
    }

    // more methods
    . . .

    // instance fields

    private String name;
    private double salary;
    private Date hireDay;
 }
```

这里将这个类的实现细节分成以下几个部分，并分别在稍后的几节中给予介绍。下面先看看例4-2，它显示了一个使用Employee类的程序代码。

在这个程序中，构造了一个Employee数组，并填入了三个雇员对象：

```
Employee[] staff = new Employee[3];

staff[0] = new Employee("Carl Cracker", . . .);
staff[1] = new Employee("Harry Hacker", . . .);
staff[2] = new Employee("Tony Tester", . . .);
```

接下来，利用Employee类的raiseSalary方法将每个雇员的薪水提高5%：

```
for (Employee e : staff)
   e.raiseSalary(5);
```

最后，调用getName方法、getSalary方法和getHireDay方法将每个雇员的信息打印出来：

```
for (Employee e : staff)
   System.out.println("name=" + e.getName()
      + ",salary=" + e.getSalary()
      + ",hireDay=" + e.getHireDay());
```

注意，在这个示例程序中包含两个类：一个Employee类；一个带有public访问修饰符的EmployeeTest类。EmployeeTest类包含了main方法，其中使用了前面介绍的指令。

源文件名是EmployeeTest.java，这是因为文件名必须与public类的名字相匹配。在一个源文件中，只能有一个公有类，但可以有任意数目的非公有类。

接下来，当编译这段源代码的时候，编译器将在目录下创建两个类文件：EmployeeTest.class和Employee.class。

将程序中包含main方法的类名字提供给字节码解释器，以便启动这个程序：

```
java EmployeeTest
```

字节码解释器开始运行EmployeeTest类的main方法中的代码。在这段代码中，先后构造了三个新Employee对象，并显示它们的状态。

例4-2 EmployeeTest.java

```
import java.util.*;

/**
 * This program tests the Employee class.
 * @version 1.11 2004-02-19
 * @author Cay Horstmann
 */
public class EmployeeTest
{
   public static void main(String[] args)
   {
      // fill the staff array with three Employee objects
      Employee[] staff = new Employee[3];

      staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
      staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
      staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);

      // raise everyone's salary by 5%
      for (Employee e : staff)
         e.raiseSalary(5);

      // print out information about all Employee objects
      for (Employee e : staff)
         System.out.println("name=" + e.getName() + ",salary=" + e.getSalary() + ",hireDay="
               + e.getHireDay());
   }
}

class Employee
{
   public Employee(String n, double s, int year, int month, int day)
   {
      name = n;
      salary = s;
      GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
      // GregorianCalendar uses 0 for January
      hireDay = calendar.getTime();
   }

   public String getName()
   {
      return name;
   }

   public double getSalary()
   {
      return salary;
   }

   public Date getHireDay()
```

```
	{
		return hireDay;
	}

	public void raiseSalary(double byPercent)
	{
		double raise = salary * byPercent / 100;
		salary += raise;
	}

	private String name;
	private double salary;
	private Date hireDay;
}
```

4.3.2 多个源文件的使用

在例4-2中，一个源文件包含了两个类。许多程序员习惯于将每一个类存在一个单独的源文件中。例如，将Employee类存放在文件Employee.java中，将EmployeeTest类存放在文件EmployeeTest.java中。

如果喜欢这样组织文件，将可以有两种编译源程序的方法。一种是使用通配符调用Java编译器：

```
javac Employee*.java
```

于是，所有与通配符匹配的源文件都将被编译成类文件。或者键入下列命令：

```
javac EmployeeTest.java
```

读者可能会感到惊讶，使用第二种方式，并没有显式地编译Employee.java。然而，当Java编译器发现EmployeeTest.java使用了Employee类时会查找名为Employee.class的文件。如果没有找到这个文件，就会自动地搜索Employee.java，然后，对它进行编译。更重要的是：如果Employee.java版本较已有的Employee.class文件版本新，Java编译器就会自动地重新编译这个文件。



注释：如果熟悉UNIX的“make”工具（或者是Windows中的“nmake”等工具），就可以认为Java编译器内置了“make”功能。

4.3.3 解析Employee类

下面对Employee类进行一下剖析。首先从这个类的方法开始。通过查看源代码会发现，这个类包含一个构造器和4个方法：

```
public Employee(String n, double s, int year, int month, int day)
public String getName()
public double getSalary()
public Date getHireDay()
public void raiseSalary(double byPercent)
```

这个类的所有方法都被标记为public。关键字public意味着任何类的任何方法都可以调用这些方法（共有4种访问级别，将在本章稍后和下一章中介绍）。

接下来，需要注意在Employee类的实例中有三个实例域用来存放将要操作的数据：

```
private String name;
private double salary;
private Date hireDay;
```

关键字private确保只有Employee类自身的方法能够访问这些实例域，而其他类的方法不能够读写这些域。

 **注释**：可以用public标记实例域，但这是一种极为不提倡的做法。public数据域允许程序中的任何方法对其进行读取和修改。这就完全破坏了封装。任何类的任何方法都可以修改public域，在我们的经历中，某些代码将使用这种存取权限，而这并不我们所希望的，因此，这里强烈建议将实例域标记为private。

最后，请注意，有两个实例域本身就是对象：name域是String类对象，hireDay域是Date类对象。这种情形十分常见：类通常包括类型属于某个类的实例域。

4.3.4 从构造器开始

下面先看看Employee类的构造器：

```
public Employee(String n, double s, int year, int month, int day)
{
   name = n;
   salary = s;
   GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
   hireDay = calendar.getTime();
}
```

已经看到，构造器与类同名。在构造Employee类的对象时，构造器被运行，以便将实例域初始化为所希望的状态。

例如，当使用下面这条代码创建Employee类实例时：

```
new Employee("James Bond", 100000, 1950, 1, 1);
```

将会把实例域设置为：

```
name = "James Bond";
salary = 100000;
hireDay = January 1, 1950;
```

构造器与其他的方法有一个重要的不同。构造器总是伴随着new操作符的执行被调用，而不能对一个已经存在的对象调用构造器来达到重新设置实例域的目的。例如，

```
james.Employee("James Bond", 250000, 1950, 1, 1); // ERROR
```

将产生编译错误。

本章稍后，还会更加详细地介绍有关构造器的内容。现在只需要记住：

- 构造器与类同名
- 每个类可以有一个以上的构造器
- 构造器可以有0个、1个或1个以上的参数
- 构造器没有返回值
- 构造器总是伴随着new操作一起调用

C++注释：Java构造器的工作方式与C++一样。但是，要记住所有的Java对象都是在堆中构造的，构造器总是伴随着new操作符一起使用。C++程序员最易犯的错误就是忘记new操作符：

```
Employee number007("James Bond", 100000, 1950, 1, 1);
   // C++, not Java
```

这条语句在C++中能够正常运行，但在Java中却不行。

警告：请注意，不要在构造器中定义与实例域重名的局部变量。例如，下面的构造器将无法设置salary。

```
public Employee(String n, double s, . . .)
{
   String name = n; // ERROR
   double salary = s; // ERROR
   . . .
}
```

这个构造器声明了局部变量name和salary。这些变量只能在构造器内部访问。这些变量屏蔽了同名的实例域。有些程序设计者（例如，本书的作者）常常不假思索地写出这类代码，因为他们习惯于增加这类数据类型。这种错误很难被检查出来，因此，必须注意在所有的方法中不要命名与实例域同名的变量。

4.3.5 隐式参数与显式参数

方法用于操作对象以及存取它们的实例域。例如，方法：

```
public void raiseSalary(double byPercent)
{
   double raise = salary * byPercent / 100;
   salary += raise;
}
```

将调用这个方法的对象的salary实例域设置为新值。看看下面这个调用：

```
number007.raiseSalary(5);
```

它的结果将number007.salary域的值增加5%。具体地说，这个调用将执行下列指令：

```
double raise = number007.salary * 5 / 100;
number007.salary += raise;
```

raiseSalary方法有两个参数。第一个参数被称为隐式（implicit）参数，是出现在方法名前的Employee类对象。第二个参数位于方法名后面括号中的数值，这是一个显式（explicit）参数。

已经看到，显式参数是明显地列在方法声明中的显示参数，例如double byPercent。隐式参数没有出现在方法声明中。

在每一个方法中，关键字this表示隐式参数。如果需要的话，可以用下列方式编写raiseSalary方法：

```
public void raiseSalary(double byPercent)
{
   double raise = this.salary * byPercent / 100;
   this.salary += raise;
}
```

有些程序员更偏爱这样的风格，因为这样可以将实例域与局部变量明显地区分开来。

C++注释：在C++中，通常在类的外面定义方法：

```
void Employee::raiseSalary(double byPercent) // C++, not Java
{
   . . .
}
```

如果在类的内部定义方法，这个方法将自动地成为内联（inline）方法。

```
class Employee
{
   . . .
   int getName() { return name; } // inline in C++
}
```

在Java程序设计语言中，所有的方法都必须在类的内部定义，但并不表示它们是内联方法。是否将某个方法设置为内联方法是Java虚拟机的任务。即时编译器会监视调用那些简洁、经常被调用、没有被重载以及可优化的方法。

4.3.6 封装的优点

最后，再仔细地看一下非常简单的getName方法、getSalary方法和getHireDay方法。

```
public String getName()
{
   return name;
}

public double getSalary()
{
   return salary;
}

public Date getHireDay()
{
   return hireDay;
}
```

这些都是典型的访问器方法。由于它们只返回实例域值，因此又被称为域访问器。

将name、salary和hireDay域标记为public，以此来取代独立的访问器方法会不会更容易些呢？

关键在于name是一个只读域。一旦在构造器中设置完毕，就没有任何一个办法可以对它进行修改，这样来确保name域不会受到外界的干扰。

虽然salary不是只读域，但是它只能用raiseSalary方法修改。特别是一旦这个域值出现了错误，只要调试这个方法就可以了。如果salary域是public的，破坏这个域值的捣乱者有可能会出没在任何地方。

在有些时候，需要获得或设置实例域的值。因此，应该提供下面三项内容：

- 一个私有的数据域；
- 一个公有的域访问器方法；
- 一个公有的域更改器方法。

这样做要比提供一个简单的公有数据域复杂些，但是却有着下列明显的好处：

1）可以改变内部实现，除了该类的方法之外，不会影响其他代码。

例如，如果将存储名字的域改为：

```
String firstName;
String lastName;
```

那么getName方法可以改为返回

```
firstName + " " + lastName
```

对于这点改变，程序的其他部分完全不可见。

当然，为了进行新旧数据表示之间的转换，访问器方法和更改器方法有可能需要做许多工作。但是，这将为我们带来了第二点好处。

2）更改器方法可以执行错误检查，然而直接对域进行赋值将不会进行这些处理。

例如，setSalary方法可以检查薪金是否小于0。

警告：注意不要编写返回引用可变对象的访问器方法。在Employee类中就违反了这个设计原则，其中的getHireDay方法返回了一个Date类对象：

```
class Employee
{
   . . .
   public Date getHireDay()
   {
      return hireDay;
   }
   . . .
   private Date hireDay;
}
```

这样会破坏封装性！请看下面这段代码：

```
Employee harry = . . .;
Date d = harry.getHireDay();
double tenYearsInMilliSeconds = 10 * 365.25 * 24 * 60 * 60 * 1000;
d.setTime(d.getTime() - (long) tenYearsInMilliSeconds)
// let's give Harry ten years added seniority
```

出错的原因很微妙。d和harry.hireDay引用同一个对象（请参见图4-5）。对d调用更改器方法就可以自动地改变这个雇员对象的私有状态！

如果需要返回一个可变对象的引用，应该首先对它进行克隆（clone）。对象克隆是指存放在另一个位置上的对象副本。有关对象克隆的详细内容将在第6章中讨论。下面是修改后的代码：

```
class Employee
{
   . . .
   public Date getHireDay()
   {
      return (Date) hireDay.clone();
   }
   . . .
}
```

凭经验可知，如果需要返回一个可变数据域的拷贝，就应该使用克隆。

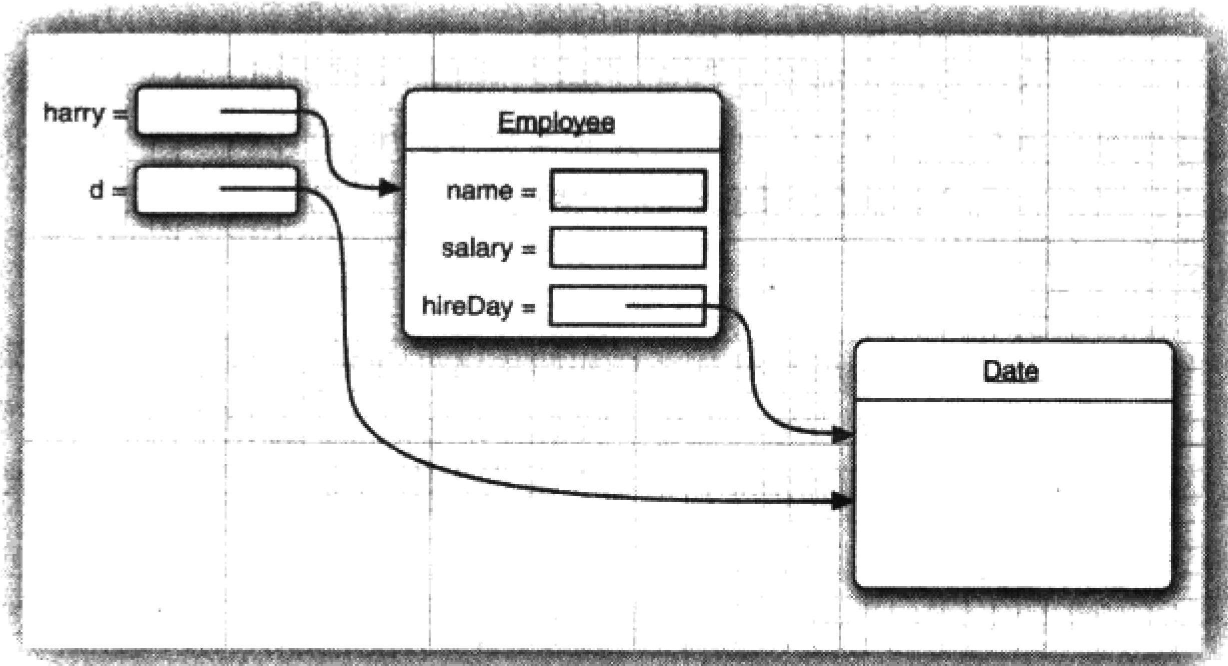


图4-5 返回可变数据域的引用

4.3.7 基于类的访问权限

从前面已经知道，方法可以访问所调用对象的私有数据。一个方法可以访问所属类的所有对象的私有数据，这令很多人感到奇怪！例如，下面看一下用来比较两个雇员的equals方法。

```
class Employee
{
   . . .
   boolean equals(Employee other)
   {
      return name.equals(other.name);
   }
}
```

典型的调用方式是

```
if (harry.equals(boss)) . . .
```

这个方法访问harry的私有域，这点并不会引发奇怪。然而，还访问boss的私有域。这是合法的，其原因是boss是Employee类对象，而Employee类的方法可以访问Employee类的任何一个对象的私有域。

C++注释：C++也有同样的原则。方法可以访问所属类的私有特性（feature），而不仅限于访问隐式参数的私有特性。

4.3.8 私有方法

在实现一个类时，由于公有数据非常危险，所以应该将所有的数据域都设置为私有的。然而，方法又应该如何设计呢？尽管绝大多数方法都被设计为公有的，但在某些特殊情况下，也可能将它们设计为私有的。有时，可能希望将一个计算代码划分成若干个独立的辅助方法。通常，这些辅助方法不应该成为公有接口的一部分，这是由于它们往往与当前的实现机制非常紧密，或者需要一个特别的协议以及一个特别的调用次序。这样的方法最好被设计为private的。

在Java中，为了实现一个私有的方法，只需要将关键字public改为private即可。

对于私有方法，如果改用其他方法实现相应的操作，则不必保留原有的方法。如果数据的表达方式发生了变化，这个方法可能会变得难以实现，或者不再需要。然而，只要方法是私有的，类的设计者就可以确信：它不会被外部的其他类操作调用，可以将其删去。如果方法是公有的，就不能将其删去，因为其他的代码很可能调用它。

4.3.9 Final实例域

可以将实例域定义为final。构建对象时必须初始化这样的域。也就是说，必须确保在每一个构造器执行之后，这个域的值被设置，并且在后面的操作中，不能够再对它进行修改。例如，可以将Employee类中的name域声明为final，因为在对象构建之后，这个值不会再被修改，即没有setName方法。

```
class Employee
{
   . . .
   private final String name;
}
```

final修饰符大都应用于*基本数据*（primitive）类型域，或*不可变*（immutable）类的域（如果类中的每个方法都不会改变其对象，这种类就是不可变的类。例如，String类就是一个不可变的类）。对于可变的类，使用final修饰符可能会对读者造成混乱。例如，

```
private final Date hiredate;
```

仅仅意味着存储在hiredate变量中的对象引用在对象构造之后不能被改变，而并不意味着hiredate 对象是一个常量。任何方法都可以对hiredate引用的对象调用setTime更改器。

4.4 静态域与静态方法

在前面给出的示例程序中，main方法都被标记为static修饰符。下面讨论一下这个修饰符的含义。

4.4.1 静态域

如果将域定义为static，每个类中只有一个这样的域。而每一个对象对于所有的实例域却都有自己的一份拷贝。例如，假定需要给每一个雇员赋予惟一的标识码。这里给Employee类添加一个实例域id和一个静态域nextId：

```
class Employee
{
   . . .
   private int id;
   private static int nextId = 1;
}
```

现在，每一个雇员对象都有一个自己的id域，但这个类的所有实例将共享一个nextId域。换句话说，如果有1000个Employee类的对象，则有1000个实例域id。但是，只有一个静态域nextId。即使没有一个雇员对象，静态域nextId也存在。它属于类，而不属于任何独立的对象。

 **注释**：*在绝大多数的面向对象程序设计语言中，静态域被称为类域。术语“static”只是沿用了C++的叫法，并无实际意义。*

下面实现一个简单的方法：

```
public void setId()
{
   id = nextId;
   nextId++;
}
```

假定为harry设定雇员标识码：

```
harry.setId();
```

harry的id域被设置为静态域nextId当前的值，并且静态域nextId的值加1：

```
harry.id = Employee.nextId;
Employee.nextId++;
```

4.4.2 静态常量

静态变量使用得比较少，但静态常量却使用得比较多。例如，在Math类中定义了一个静态常量：

```
public class Math
{
   . . .
   public static final double PI = 3.14159265358979323846;
   . . .
}
```

在程序中，可以采用Math.PI的形式获得这个常量。

如果关键字static被省略，PI就变成了Math类的一个实例域。需要通过Math类的对象访问PI，并且每一个Math对象都有它自己的一份PI拷贝。

另一个多次使用的静态常量是System.out。它在System类中声明：

```
public class System
{
   . . .
   public static final PrintStream out = . . .;
   . . .
}
```

前面曾经提到过，由于每个类对象都可以对公有域进行修改，所以，最好不要将域设计为public。然而，公有常量（即final域）却没问题。因为out被声明为final，所以，不允许再将其他打印流赋给它：

```
System.out = new PrintStream(. . .); // ERROR--out is final
```

✔ **注释**：如果查看一下System类，就会发现有一个setOut方法，它可以将System.out设置为不同的流。读者可能会感到奇怪，为什么这个方法可以修改final变量的值。原因在于，setOut方法是一个本地方法，而不是用Java语言实现的。本地方法可以绕过Java语言的存取控制机制。这是一种特殊的方法，在自己编写程序时，不应该这样处理。

4.4.3 静态方法

静态方法是一种不能向对象实施操作的方法。例如，Math类的pow方法就是一个静态方法。表达式

```
Math.pow(x, a)
```

计算幂x^a。在运算时，不使用任何Math对象。换句话说，没有隐式的参数。

可以认为静态方法是没有this参数的方法（在一个非静态的方法中，this参数表示这个方法的隐式参数）。

因为静态方法不能操作对象，所以不能在静态方法中访问实例域。但是，静态方法可以访问自身类中的静态域。下面是使用这种静态方法的一个示例：

```
public static int getNextId()
{
   return nextId; // returns static field
}
```

可以通过类名调用这个方法：

```
int n = Employee.getNextId();
```

这个方法可以省略关键字static吗？答案是肯定的。但是，需要通过Employee类对象的引用调用这个方法。

注释：可以使用对象调用静态方法。例如，如果harry是一个Employee对象，可以用harry.getNextId( )代替Employee.getnextId( )。不过，这种方式很容易造成混淆，其原因是getNextId方法计算的结果与harry毫无关系。我们建议使用类名，而不是对象来调用静态方法。

在下面两种情况下使用静态方法：

- 一个方法不需要访问对象状态，其所需参数都是通过显式参数提供（例如：Math.pow）。
- 一个方法只需要访问类的静态域（例如：Employee.getNextId）。

C++注释：Java中的静态域与静态方法在功能上与C++相同。但是，语法书写上却稍有所不同。在C++中，使用::操作符访问自身作用域之外的静态域和静态方法，如Math::PI。术语“static”有一段不寻常的历史。起初，C引入关键字static是为了表示退出一个块后依然存在的局部变量。在这种情况下，术语“static”是有意义的：变量一直存在，当再次进入该块时仍然存在。随后，static在C中有了第二种含义，表示不能被其他文件访问的全局变量和函数。为了避免引入一个新的关键字，关键字static被重用了。最后，C++第三次重用了这个关键字，与前面赋予的含义完全不一样，这里将其解释为：属于类且不属于类对象的变量和函数。这个含义与Java相同。

4.4.4 Factory方法

静态方法还有一种常见的用途。NumberFormat类使用factory方法产生不同风格的格式对象。

```
NumberFormat currencyFormatter = NumberFormat.getCurrencyInstance();
NumberFormat percentFormatter = NumberFormat.getPercentInstance();
double x = 0.1;
System.out.println(currencyFormatter.format(x)); // prints $0.10
System.out.println(percentFormatter.format(x)); // prints 10%
```

为什么NumberFormat类不利用构造器完成这些操作呢？这主要有两个原因：

- 无法命名构造器。构造器的名字必须与类名相同。但是，这里希望将得到的货币实例和

百分比实例采用不用的名字。

- 当使用构造器时，无法改变所构造的对象类型。而Factory方法将返回一个DecimalFormat类对象，这是NumberFormat的子类（有关继承的详细内容请参看第5章）。

4.4.5 Main方法

需要注意，不需要使用对象调用静态方法。例如，不需要构造Math类对象就可以调用Math.pow。

同理，main方法也是一个静态方法。

```
public class Application
{
   public static void main(String[] args)
   {
      // construct objects here
      . . .
   }
}
```

main方法不对任何对象进行操作。事实上，在启动程序时还没有任何一个对象。静态的main方法将执行并创建程序所需要的对象。

提示：每一个类可以有一个main方法。这是一个常用于对类进行单元测试的技巧。例如，可以在Employee类中添加一个main方法：

```
class Employee
{
   public Employee(String n, double s, int year, int month, int day)
   {
      name = n;
      salary = s;
      GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
      hireDay = calendar.getTime();
   }
   . . .
   public static void main(String[] args) // unit test
   {
      Employee e = new Employee("Romeo", 50000, 2003, 3, 31);
      e.raiseSalary(10);
      System.out.println(e.getName() + " " + e.getSalary());
   }
   . . .
}
```

如果想要独立地测试Employee类，只需要执行

```
java Employee
```

如果雇员类是大型应用程序的一部分，就可以使用下面这条语句运行程序

```
java Application
```

并且Employee类的main方法永远不会被执行。

例4-3中的程序包含了Employee类的一个简单版本，其中有一个静态域nextId和一个静态方法getNextId。这里将三个Employee对象写入数组，然后打印雇员信息。最后，打印出下一个可

用的员工标识码来作为对静态方法使用的演示。

需要注意，Employee类也有一个静态的main方法用于单元测试。试试运行

```
java Employee
```

和

```
java StaticTest
```

执行两个main方法。

例4-3 StaticTest.java

```java
/**
 * This program demonstrates static methods.
 * @version 1.01 2004-02-19
 * @author Cay Horstmann
 */
public class StaticTest
{
   public static void main(String[] args)
   {
      // fill the staff array with three Employee objects
      Employee[] staff = new Employee[3];

      staff[0] = new Employee("Tom", 40000);
      staff[1] = new Employee("Dick", 60000);
      staff[2] = new Employee("Harry", 65000);

      // print out information about all Employee objects
      for (Employee e : staff)
      {
         e.setId();
         System.out.println("name=" + e.getName() + ",id=" + e.getId() + ",salary="
               + e.getSalary());
      }

      int n = Employee.getNextId(); // calls static method
      System.out.println("Next available id=" + n);
   }
}

class Employee
{
   public Employee(String n, double s)
   {
      name = n;
      salary = s;
      id = 0;
   }

   public String getName()
   {
      return name;
   }

   public double getSalary()
   {
      return salary;
```

```
   }

   public int getId()
   {
      return id;
   }

   public void setId()
   {
      id = nextId; // set id to next available id
      nextId++;
   }

   public static int getNextId()
   {
      return nextId; // returns static field
   }

   public static void main(String[] args) // unit test
   {
      Employee e = new Employee("Harry", 50000);
      System.out.println(e.getName() + " " + e.getSalary());
   }

   private String name;
   private double salary;
   private int id;
   private static int nextId = 1;
}
```

4.5 方法参数

首先回顾一下在程序设计语言中有关将参数传递给方法（或函数）的一些专业术语。值调用（call by value）表示方法接收的是调用者提供的值。而引用调用（call by reference）表示方法接收的是调用者提供的变量地址。一个方法可以修改传递引用所对应的变量值，而不能修改传递值调用所对应的变量值。“……调用”（call by）是一个标准的计算机科学术语，它用来描述各种程序设计语言中方法参数的传递方式（事实上，以前还有称（call by name），Algol程序设计语言是最古老的高级程序设计语言之一，它使用的就是这种参数传递方式。不过，对于今天，这种传递方式已经成为历史）。

Java程序设计语言总是采用值调用。也就是说，方法得到的是所有参数值的一个拷贝，特别是，方法不能修改传递给它的任何参数变量的内容。

例如，考虑下面的调用：

```
double percent = 10;
harry.raiseSalary(percent);
```

不必理睬这个方法的具体实现，在方法调用之后，percent的值还是10。

下面再仔细地研究一下这种情况。假定一个方法试图将一个参数值增加至3倍：

```
public static void tripleValue(double x) // doesn't work
{
   x = 3 * x;
}
```

然后调用这个方法：

```
double percent = 10;
tripleValue(percent);
```

可以看到，无论怎样，调用这个方法之后，percent的值还是10。下面看一下具体的执行过程：

1）x被初始化为percent值的一个拷贝（也就是10）。

2）x被乘以3后等于30。但是percent仍然是10（如图4-6所示）。

3）这个方法结束之后，参数变量x不再使用。

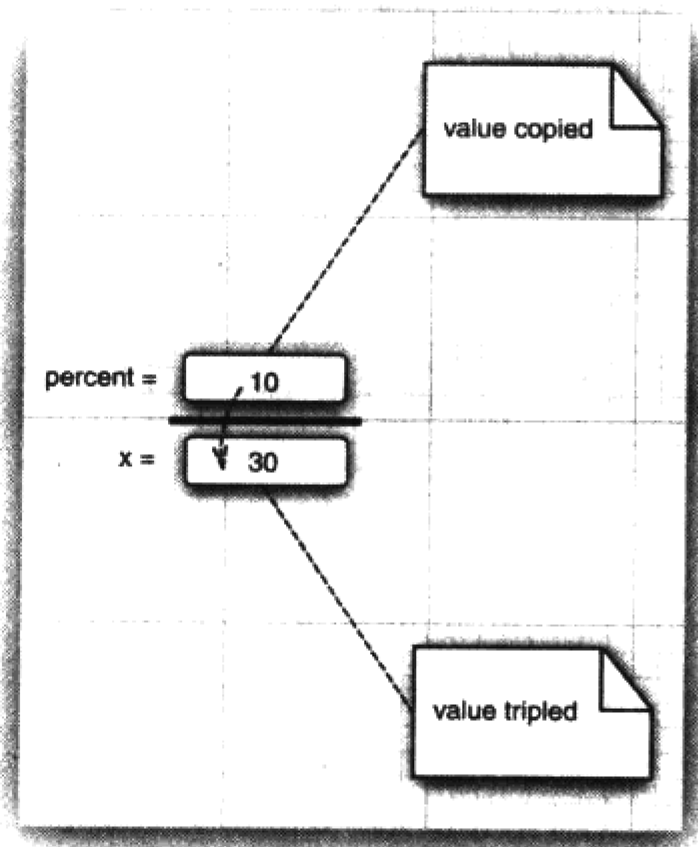


图4-6 对值参数的修改没有保留下来

然而，方法参数共有两种类型：

- 基本数据类型(数字、布尔值)。
- 对象引用。

读者已经看到，一个方法不可能修改一个基本数据类型的参数。而对象引用作为参数就不同了，可以很容易地利用下面这个方法实现将一个雇员的薪金提高两倍的操作：

```
public static void tripleSalary(Employee x) // works
{
   x.raiseSalary(200);
}
```

当调用

```
harry = new Employee(. . .);
tripleSalary(harry);
```

时，具体的执行过程为：

1）x被初始化为harry值的拷贝，这里是一个对象的引用。

2）raiseSalary方法应用于这个对象引用。x和harry同时引用的那个Employee对象的薪金提

高了200%。

3）方法结束后，参数变量x不再使用。当然，对象变量harry继续引用那个薪金增至3倍的雇员对象（如图4-7所示）。

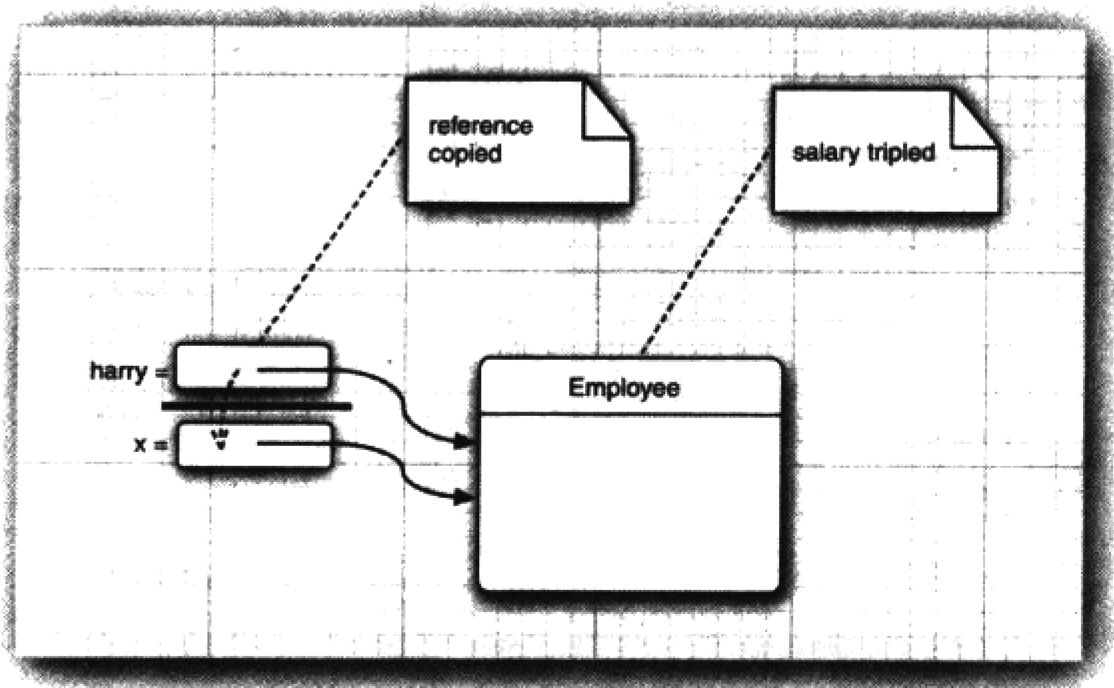


图4-7 对对象参数的修改保留了下来

读者已经看到，实现一个改变对象参数状态的方法并不是一件难事。理由很简单，方法得到的是对象引用的拷贝，对象引用及其他的拷贝同时引用同一个对象。

很多程序设计语言（特别是，C++和Pascal）提供了两种参数传递的方式：值调用和引用调用。有些程序员（甚至本书的作者）认为Java 程序设计语言对对象采用的是引用调用，实际上，这种理解是不对的。由于这种误解具有一定的普遍性，所以下面给出一个反例来详细地阐述一下这个问题。

首先，编写一个交换两个雇员对象的方法：

```
public static void swap(Employee x, Employee y) // doesn't work
{
   Employee temp = x;
   x = y;
   y = temp;
}
```

如果Java程序设计语言对对象采用的是引用调用，那么这个方法就应该能够实现交换数据的效果：

```
Employee a = new Employee("Alice", . . .);
Employee b = new Employee("Bob", . . .);
swap(a, b);
// does a now refer to Bob, b to Alice?
```

但是，方法并没有改变存储在变量a和b中的对象引用。swap方法的参数x和y被初始化为两个对象引用的拷贝，这个方法交换的是这两个拷贝。

```
// x refers to Alice, y to Bob
Employee temp = x;
x = y;
y = temp;
// now x refers to Bob, y to Alice
```

最终，白费力气。在方法结束时参数变量x和y被丢弃了。原来的变量a和b仍然引用这个方法调用之前所引用的对象（如图4-8所示）。

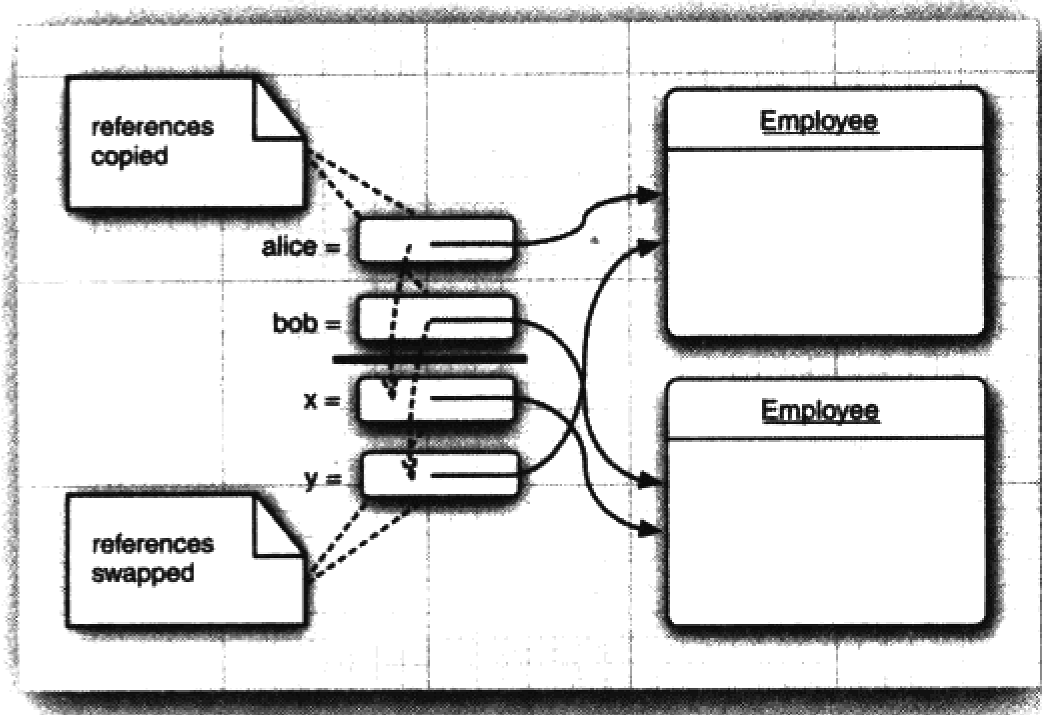


图4-8 交换对象参数的结果没有被保下来

这个过程说明：Java程序设计语言对对象采用的不是引用调用，实际上，对象引用进行的是值传递。

下面总结一下在Java程序设计语言中，方法参数的使用情况：

- 一个方法不能修改一个基本数据类型的参数（即数值型和布尔型）。
- 一个方法可以改变一个对象参数的状态。
- 一个方法不能实现让对象参数引用一个新的对象。

例4-4中的程序给出了相应的演示。在这个程序中，首先试图将一个值参数的值提高两倍，但没有成功：

```
Testing tripleValue:
Before: percent=10.0
End of method: x=30.0
After: percent=10.0
```

随后，成功地将一个雇员的薪金提高了两倍：

```
Testing tripleSalary:
Before: salary=50000.0
End of method: salary=150000.0
After: salary=150000.0
```

方法结束之后，harry引用的对象状态发生了改变。这是因为这个方法可以通过对象引用的拷贝修改所引用的对象状态。

最后，程序演示了swap方法的失败效果：

```
Testing swap:
Before: a=Alice
Before: b=Bob
End of method: x=Bob
```

```
End of method: y=Alice
After: a=Alice
After: b=Bob
```

可以看出，参数变量x和y被交换了，但是变量a和b没有受到影响。

C++注释：C++有值调用和引用调用。引用参数标有&符号。例如，可以轻松地实现void tripleValue(double& x) 方法或void swap(Employee& x, Employee& y) 方法实现修改它们的引用参数的目的。

例4-4 ParamTest.java

```java
/**
 * This program demonstrates parameter passing in Java.
 * @version 1.00 2000-01-27
 * @author Cay Horstmann
 */
public class ParamTest
{
   public static void main(String[] args)
   {
      /*
       * Test 1: Methods can't modify numeric parameters
       */
      System.out.println("Testing tripleValue:");
      double percent = 10;
      System.out.println("Before: percent=" + percent);
      tripleValue(percent);
      System.out.println("After: percent=" + percent);

      /*
       * Test 2: Methods can change the state of object parameters
       */
      System.out.println("\nTesting tripleSalary:");
      Employee harry = new Employee("Harry", 50000);
      System.out.println("Before: salary=" + harry.getSalary());
      tripleSalary(harry);
      System.out.println("After: salary=" + harry.getSalary());

      /*
       * Test 3: Methods can't attach new objects to object parameters
       */
      System.out.println("\nTesting swap:");
      Employee a = new Employee("Alice", 70000);
      Employee b = new Employee("Bob", 60000);
      System.out.println("Before: a=" + a.getName());
      System.out.println("Before: b=" + b.getName());
      swap(a, b);
      System.out.println("After: a=" + a.getName());
      System.out.println("After: b=" + b.getName());
   }

   public static void tripleValue(double x) // doesn't work
   {
      x = 3 * x;
      System.out.println("End of method: x=" + x);
```

```
    }

    public static void tripleSalary(Employee x) // works
    {
       x.raiseSalary(200);
       System.out.println("End of method: salary=" + x.getSalary());
    }

    public static void swap(Employee x, Employee y)
    {
       Employee temp = x;
       x = y;
       y = temp;
       System.out.println("End of method: x=" + x.getName());
       System.out.println("End of method: y=" + y.getName());
    }
}

class Employee // simplified Employee class
{
    public Employee(String n, double s)
    {
       name = n;
       salary = s;
    }

    public String getName()
    {
       return name;
    }

    public double getSalary()
    {
       return salary;
    }

    public void raiseSalary(double byPercent)
    {
       double raise = salary * byPercent / 100;
       salary += raise;
    }

    private String name;
    private double salary;
}
```

4.6 对象构造

前面已经学会了编写简单的构造器，以便对定义的对象进行初始化。但是，由于对象构造非常重要，所以Java提供了多种编写构造器的方式。下面将详细地介绍一下。

4.6.1 重载

从前面可以看到，GregorianCalendar类有多个构造器。可以使用：

```
GregorianCalendar today = new GregorianCalendar();
```

或者

```
GregorianCalendar deadline = new GregorianCalendar(2099, Calendar.DECEMBER, 31);
```

这种特征叫做重载（overloading）。如果多个方法（比如，GregorianCalendar构造器方法）有相同的名字、不同的参数，便产生了重载。编译器必须挑选出具体执行哪个方法，它通过用各个方法给出的参数类型与特定方法调用所使用的值类型进行匹配来挑选出相应的方法。如果编译器找不到匹配的参数，或者找出多个可能的匹配，就会产生编译时错误（这个过程被称为重载解析（overloading resolution）。）



注释：Java允许重载任何方法，而不只是构造器方法。因此，要完整地描述一个方法，需要指出方法名以及参数类型。这叫做方法的签名（signature）。例如，String类有4个称为indexOf的公有方法。它们的签名是

```
indexOf(int)
indexOf(int, int)
indexOf(String)
indexOf(String, int)
```

返回类型不是方法签名的一部分。也就是说，不能有两个名字相同、参数类型也相同却返回不同类型值的方法。

4.6.2 默认域初始化

如果在构造器中没有显式地给域赋予初值，那么就会被自动地赋为默认值：数值为0、布尔值为flase、对象引用为null。然而，只有缺少程序设计经验的人才会这样做。确实，如果不明确地对域进行初始化，就会影响程序代码的可读性。



注释：这是域与局部变量的主要不同点。必须明确地初始化方法中的局部变量。但是，如果没有初始化类中的域，将会被初始化为默认值（0、false或null）。

例如，仔细看一下Employee类。假定没有在构造器中对某些域进行初始化，就会默认地将salary域初始化为0，将name、hireDay域初始化为null。

但是，这并不是一种良好的编程习惯。如果此时调用getName方法或getHireDay方法，则会得到一个null引用，这应该不是我们所希望的结果：

```
Date h = harry.getHireDay();
calendar.setTime(h); // throws exception if h is null
```

4.6.3 默认构造器

所谓默认构造器是指没有参数的构造器。例如，Employee类的默认构造器：

```
public Employee()
{
   name = "";
   salary = 0;
   hireDay = new Date();
}
```

如果在编写一个类时没有编写构造器，那么系统就会提供一个默认构造器。这个默认构造

器将所有的实例域设置为默认值。于是，实例域中的数值型数据设置为0、布尔型数据设置为false、所有对象变量将设置为null。

如果类中提供了至少一个构造器，但是没有提供默认的构造器，则在构造对象时如果没有提供构造参数就会被视为不合法。例如，在例4-2中的Employee类提供了一个简单的构造器：

```
Employee(String name, double salary, int y, int m, int d)
```

对于这个类，构造默认的雇员属于不合法。也就是，调用

```
e = new Employee();
```

将会产生错误。

警告：请记住，仅当类没有提供任何构造器的时候，系统才会提供一个默认的构造器。如果在编写类的时候，给出了一个构造器，哪怕是很简单的，要想让这个类的用户能够采用下列方式构造实例：

```
new ClassName()
```

就必须提供一个默认的构造器（即不带参数的构造器）。当然，如果希望所有域被赋予默认值，可以采用下列格式：

```
public ClassName()
{
}
```

4.6.4 显式域初始化

由于类的构造器方法可以重载，所以可以采用多种形式设置类的实例域的初始状态。确保不管怎样调用构造器，每个实例域都可以被设置为一个有意义的初值。这是一种很好的设计习惯。

可以在类定义中，直接将一个值赋给任何域。例如：

```
class Employee
{
   . . .
   private String name = "";
}
```

在执行构造器之前，先执行赋值操作。当一个类的所有构造器都希望把相同的值赋予某个特定的实例域时，这种方式特别有用。

初始值不一定是常量。在下面的例子中，可以调用方法对域进行初始化。仔细看一下Employee类，其中每个雇员有一个id域。可以使用下列方式进行初始化：

```
class Employee
{
   . . .
   static int assignId()
   {
      int r = nextId;
      nextId++;
      return r;
   }
   . . .
   private int id = assignId();
}
```

C++注释：在C++中，不能直接初始化实例域。所有的域必须在构造器中设置。但是，有一个特殊的初始化器列表语法，如下所示：

```
Employee::Employee(String n, double s, int y, int m, int d) // C++
:  name(n),
   salary(s),
   hireDay(y, m, d)
{
}
```

C++使用这种特殊的语法来调用域构造器。在Java中没有这种必要，因为对象没有子对象，只有指向其他对象的指针。

4.6.5 参数名

在编写很小的构造器时（这是十分常见的），常常在参数命名上出现错误。通常，参数用单个字符命名：

```
public Employee(String n, double s)
{
   name = n;
   salary = s;
}
```

但这样做有一个缺陷：只有阅读代码才能够了解参数n和参数s的含义。于是，有些程序员在每个参数前面加上一个前缀“a”：

```
public Employee(String aName, double aSalary)
{
   name = aName;
   salary = aSalary;
}
```

这样很清晰。每一个读者一眼就能够看懂参数的含义。

还一种常用的技巧，它基于这样的事实：参数变量用同样的名字将实例域屏蔽起来。例如，如果将参数命名为salary，salary将引用这个参数，而不是实例域。但是，可以采用this.salary的形式访问实例域。回想一下，this指示隐式参数，也就是被构造的对象。下面是一个示例

```
public Employee(String name, double salary)
{
   this.name = name;
   this.salary = salary;
}
```

C++注释：在C++中，经常用下划线或某个固定的字母（一般选用m或x）作为实例域的前缀。例如，salary域可能被命名为_salary、mSalary或xSalary。Java程序员通常不这样做。

4.6.6 调用另一个构造器

关键字this引用方法的隐式参数。然而，这个关键字还有另外一个含义。

如果构造器的第一个语句形如this(...)，这个构造器将调用同一个类的另一个构造器。下面是一个典型的例子：

```
public Employee(double s)
{
   // calls Employee(String, double)
   this("Employee #" + nextId, s);
   nextId++;
}
```

当调用new Employee(60000)时，Employee(double)构造器将调用Employee(String, double)构造器。

采用这种方式使用this关键字非常有用，这样对公共的构造器代码部分只编写一次即可。

C++注释：在Java中，this引用等价于C++的this指针。但是，在C++中，一个构造器不能调用另一个构造器。在C++中，必须将抽取出的公共初始化代码编写成一个独立的方法。

4.6.7 初始化块

前面已经讲过两种初始化数据域的方法：

- 在构造器中设置值
- 在声明中赋值

实际上，Java还有第三种机制，称为*初始化块*（initialization block）。在一个类的声明中，可以包含多个代码块。只要构造类的对象，这些块就会被执行。例如，

```
class Employee
{
   public Employee(String n, double s)
   {
      name = n;
      salary = s;
   }

   public Employee()
   {
      name = "";
      salary = 0;
   }
   . . .
   private static int nextId;

   private int id;
   private String name;
   private double salary;
   . . .

   // object initialization block
   {
      id = nextId;
      nextId++;
   }
}
```

在这个示例中，无论使用哪个构造器构造对象，id域都在对象初始化块中被初始化。首先运行初始化块，然后才运行构造器的主体部分。

这种机制不是必须的，也不常见。通常，直接将初始化代码放在构造器中。

注释：即使域定义在类的后半部分，在初始化块中仍然可以为它设置值。Sun的Java编译器的某些版本错误地处理了这种情况（bug # 4459133）。这个bug在Java SE 1.4.1中已经得到修正。但是，为了避免循环定义，不要读取在后面初始化的域。具体的规则请参看Java语言规范的8.3.2.3节（http://java.sun.com/docs/books/jls）。这个规则的复杂度足以使编译器的实现者头疼，因此建议将初始化块放在域定义之后。

由于初始化数据域有多种途径，所以列出构造过程的所有路径可能相当混乱。下面是调用构造器的具体处理步骤：

1）所有数据域被初始化为默认值（0、false或null）。

2）按照在类声明中出现的次序，依次执行所有域初始化语句和初始化块。

3）如果构造器第一行调用了第二个构造器，则执行第二个构造器主体。

4）执行这个构造器的主体。

当然，应该精心地组织好初始化代码，这样有利于其他程序员的理解。例如，如果让类的构造器行为依赖于数据域声明的顺序，那就会显得很奇怪并且容易引起错误。

可以通过提供一个初始化值，或者使用一个静态的初始化块来对静态域进行初始化。前面已经介绍过第一种机制：

```
static int nextId = 1;
```

如果对类的静态域进行初始化的代码比较复杂，那么可以使用静态的初始化块。

将代码放在一个块中，并标记关键字static。下面是一个示例。其功能是将雇员ID的起始值赋予一个小于10 000的随机整数。

```
// static initialization block
static
{
   Random generator = new Random();
   nextId = generator.nextInt(10000);
}
```

在类第一次加载的时候，将会进行静态域的初始化。与实例域一样，除非将它们显式地设置成其他值，否则默认的初始值是 0、false或null。所有的静态初始化语句以及静态初始化块都将依照类定义的顺序执行。

注释：使用下面这种方式，在同伴们的面前显露一手：可以使用Java编写一个没有main方法的“Hello, World”程序。

```
public class Hello
{
   static
   {
      System.out.println("Hello, World");
   }
}
```

当用java Hello调用这个类时，这个类就被加载，静态初始化块将会打印“Hello, World”。在此之后，会得到一个“main is not defined（没有定义）”的错误信息。不过，可以在静态初始化块的尾部调用System.exit(0)避免这一缺陷。

例4-5中的程序展示了本节论述的很多特性：

- 重载构造器
- 用 this(...) 调用另一个构造器
- 默认构造器
- 对象初始化块
- 静态初始化块
- 实例域初始化

例4-5 ConstructorTest.java

```
import java.util.*;

/**
 * This program demonstrates object construction.
 * @version 1.01 2004-02-19
 * @author Cay Horstmann
 */
public class ConstructorTest
{
   public static void main(String[] args)
   {
      // fill the staff array with three Employee objects
      Employee[] staff = new Employee[3];

      staff[0] = new Employee("Harry", 40000);
      staff[1] = new Employee(60000);
      staff[2] = new Employee();

      // print out information about all Employee objects
      for (Employee e : staff)
         System.out.println("name=" + e.getName() + ",id=" + e.getId() + ",salary="
               + e.getSalary());
   }
}

class Employee
{
   // three overloaded constructors
   public Employee(String n, double s)
   {
      name = n;
      salary = s;
   }

   public Employee(double s)
   {
      // calls the Employee(String, double) constructor
      this("Employee #" + nextId, s);
   }

   // the default constructor
   public Employee()
   {
```

```
      // name initialized to ""--see below
      // salary not explicitly set--initialized to 0
      // id initialized in initialization block
   }

   public String getName()
   {
      return name;
   }

   public double getSalary()
   {
      return salary;
   }

   public int getId()
   {
      return id;
   }

   private static int nextId;

   private int id;
   private String name = ""; // instance field initialization
   private double salary;

   // static initialization block
   static
   {
      Random generator = new Random();
      // set nextId to a random number between 0 and 9999
      nextId = generator.nextInt(10000);
   }

   // object initialization block
   {
      id = nextId;
      nextId++;
   }
}
```

API java.util.Random 1.0

- `Random( )`

  构造一个新的随机数生成器。

- `int nextInt(int n)` 1.2

  返回一个0～n－1之间的随机数。

4.6.8 对象析构与finalize方法

有些面向对象的程序设计语言，特别是C++，有显式的析构器方法，其中放置一些当对象不再使用时需要执行的清理代码。在析构器中，最常见的操作是回收分配给对象的存储空间。由于Java有自动的垃圾回收器，不需要人工回收内存，所以Java不支持析构器。

当然，某些对象使用了内存之外的其他资源，例如，文件或使用了系统资源的另一个对象的句柄。在这种情况下，当资源不再需要时，将其回收和再利用将显得十分重要。

可以为任何一个类添加finalize方法。finalize方法将在垃圾回收器清除对象之前调用。在实际应用中，不要依赖于使用finalize方法回收任何短缺的资源，这是因为很难知道这个方法什么时候才能够调用。

✔ **注释**：有个名为System.runFinalizersOnExit(true)的方法能够确保finalizer方法在Java关闭前被调用。不过，这个方法并不安全，也不鼓励大家使用。有一种代替的方法是使用方法Runtime.addShutdownHook添加"关闭钩"（shutdown hook），详细内容请参看API文档。

如果某个资源需要在使用完毕后立刻被关闭，那么就需要由人工来管理。可以应用一个类似dispose或close的方法完成相应的清理操作。特别需要说明，如果一个类使用了这样的方法，当对象不再被使用时一定要调用它。

4.7 包

Java允许使用包（package）将类组织起来。借助于包可以方便地组织自己的代码，并将自己的代码与别人提供的代码库分开管理。

标准的Java类库分布在多个包中，包括java.lang、java.util和java.net等。标准的Java包具有一个层次结构。如同硬盘的目录嵌套一样，也可以使用嵌套层次组织包。所有标准的Java包都处于 java和javax包层次中。

使用包的主要原因是确保类名的惟一性。假如两个程序员不约而同地建立了Employee类。只要将这些类放置在不同的包中，就不会产生冲突。事实上，为了保证包名的绝对惟一性，Sun公司建议将公司的因特网域名（这显然是独一无二的）以逆序的形式作为包名，并且对于不同的项目使用不同的子包。例如，horstmann.com是本书作者之一注册的域名。逆序形式为com.horstmann。这个包还可以被进一步地划分成子包，如com.horstmann.corejava。

从编译器的角度来看，嵌套的包之间没有任何关系。例如，java.util包与java.util.jar包毫无关系。每一个都拥有独立的类集合。

4.7.1 类的导入

一个类可以使用所属包中的所有类，以及其他包中的公有类（public class）。我们可以采用两种方式访问另一个包中的公有类。第一种方式是在每个类名之前添加完整的包名。例如：

```
java.util.Date today = new java.util.Date();
```

这显然很令人生厌。更简单且更常用的方式是使用import语句。import语句是一种引用包含在包中的类的简明描述。一旦使用了import语句，在使用类时，就不必写出包的全名了。

可以使用import语句导入一个特定的类或者整个包。import语句应该位于源文件的顶部（但位于package语句的后面）。例如，可以使用下面这条语句导入java.util包中所有的类。

```
import java.util.*;
```

然后，就可以使用

```
Date today = new Date();
```

而无需在前面加上包前缀。还可以导入一个包中的特定类：

```
import java.util.Date;
```

java.util.*的语法比较简单，对代码的大小也没有任何负面影响。当然，如果能够明确地指出所导入的类，将会使代码的读者更加准确地知道加载了哪些类。

提示：在Eclipse中，可以使用菜单选项Source→Organize Imports。Package语句，如import java.util.*，将会自动地扩展指定的导入列表，如：import java.util.ArrayList; import java.util.Date; 这是一个十分便捷的特性。

但是，需要注意的是，只能使用星号（*）导入一个包，而不能使用import java.*或import java.*.* 导入以java为前缀的所有包。

在大多数情况下，只导入所需的包，并不必过多地理睬它们。但在发生命名冲突的时候，就不能不注意包的名字了。例如，java.util和java.sql包都有日期（Date）类。如果在程序中导入了这两个包：

```
import java.util.*;
import java.sql.*;
```

在程序使用Date类的时候，就会出现一个编译错误：

```
Date today; // ERROR--java.util.Date or java.sql.Date?
```

此时编译器无法确定程序使用的是哪一个Date类。可以采用增加一个特定的import语句来解决这个问题：

```
import java.util.*;
import java.sql.*;
import java.util.Date;
```

如果这两个Date类都需要使用，又该怎么办呢？答案是，在每个类名的前面加上完整的包名。

```
java.util.Date deadline = new java.util.Date();
java.sql.Date today = new java.sql.Date(...);
```

在包中定位类是编译器（comp iler）的工作。类文件中的字节码肯定使用完整的包名来引用其他类。

C++注释：C++程序员经常将import与#include弄混。实际上，这两者之间并没有共同之处。在C++中，必须使用#include将外部特性的声明加载进来，这是因为C++编译器无法查看任何文件的内部，除了正在编译的文件以及在头文件中明确包含的文件。Java编译器可以查看其他文件的内部，只要告诉它到哪里去查看就可以了。

在Java中，通过显式地给出包名，如java.util.Date，就可以不使用import；而在C++中，无法避免使用#include。

Import语句的惟一的好处是简捷。可以使用简短的名字而不是完整的包名来引用一个类。例如，在import java.util.*（或import java.util.Date）语句之后，可以仅仅用Date引用java.util.Date类。

在C++中，与包机制类似的是命名空间（namespace）。在Java中，package与import语句类

似于C++中的namespace和using指令（directive）。

4.7.2 静态导入

从Java SE 5.0开始，import语句不仅可以导入类，还增加了导入静态方法和静态域的功能。例如，如果在源文件的顶部，添加一条指令：

```
import static java.lang.System.*;
```

就可以使用System类的静态方法和静态域，而不必加类名前缀：

```
out.println("Goodbye, World!"); // i.e., System.out
exit(0); // i.e., System.exit
```

另外，还可以导入特定的方法或域：

```
import static java.lang.System.out;
```

实际上，是否有更多的程序员采用System.out或System.exit的简写形式，似乎是一件值得怀疑的事情。这种编写形式不利于代码的清晰度。不过，静态导入有两个实际的应用。

- 算术函数：如果对Math类使用静态导入，就可以采用更加自然的方式使用算术函数。例如，

  ```
  sqrt(pow(x, 2) + pow(y, 2))
  ```

  看起来比

  ```
  Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2))
  ```

  清晰得多。
- 笨重的常量：如果需要使用大量带有冗长名字的常量，就应该使用静态导入。例如，

  ```
  if (d.get(DAY_OF_WEEK) == MONDAY)
  ```

  看起来比

  ```
  if (d.get(Calendar.DAY_OF_WEEK) == Calendar.MONDAY)
  ```

  容易得多。

4.7.3 将类放入包中

要想将一个类放入包中，就必须将包的名字放在源文件的开头，包中定义类的代码之前。例如，例4-7中的文件Employee.java开头是这样的：

```
package com.horstmann.corejava;

public class Employee
{
   . . .
}
```

如果没有在源文件中放置package语句，这个源文件中的类就被放置在一个默认包（defaulf package）中。默认包是一个没有名字的包。在此之前，我们定义的所有类都在默认包中。

将包中的文件放到与完整的包名匹配的子目录中。例如，com.horstmann.corejava包中的所有源文件应该被放置在子目录com/horstmann/corejava（Windows中com\horstmann\corejava）中。编译器将类文件也放在相同的目录结构中。

例4-6和例4-7中的程序分放在两个包中：PackageTest类放置在默认包中；Employee类放置

在com.horstmann.corejava包中。因此，Employee.class文件必须包含在子目录com/horstmann/corejava中。换句话说，目录结构如下所示：

```
.基目录.
  ├─ PackageTest.java
  ├─ PackageTest.class
  └─ com/
      └─ horstmann/
          └─ corejava/
              ├─ Employee.java
              └─ Employee.class
```

要想编译这个程序，只需改变基目录，并运行命令

```
javac PackageTest.java
```

编译器就会自动地查找文件com/horstmann/corejava/Employee.java并进行编译。

下面看一个更加实际的例子。在这里不使用默认包，而是将类分别放在不同的包中（com.horst-mann.corejava和com.mycompany）。

```
基目录
  └─ com/
      ├─ horstmann/
      │   └─ corejava/
      │       ├─ Employee.java
      │       └─ Employee.class
      └─ mycompany/
          ├─ PayrollApp.java
          └─ PayrollApp.class
```

在这种情况下，仍然要从基目录编译和运行类，即包含com目录：

```
javac com/mycompany/PayrollApp.java
java com.mycompany.PayrollApp
```

需要注意，编译器对文件（带有文件分隔符和扩展名.java的文件）进行操作。而Java解释器加载类（带有.分隔符）。

警告：编译器在编译源文件的时候不检查目录结构。例如，假定有一个源文件开头有下列语句：

```
package com.mycompany;
```

即使这个源文件没有在子目录com/mycompany下，也可以进行编译。如果它不依赖于其他包，就不会出现编译错误。但是，最终的程序将无法运行，这是因为虚拟机找不到类文件。

例4-6 PackageTest.java

```
import com.horstmann.corejava.*;
// the Employee class is defined in that package

import static java.lang.System.*;

/**
```

```
 * This program demonstrates the use of packages.
 * @author cay
 * @version 1.11 2004-02-19
 * @author Cay Horstmann
 */
public class PackageTest
{
   public static void main(String[] args)
   {
      // because of the import statement, we don't have to use com.horstmann.corejava.Employee here
      Employee harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);

      harry.raiseSalary(5);

      // because of the static import statement, we don't have to use System.out here
      out.println("name=" + harry.getName() + ",salary=" + harry.getSalary());
   }
}
```

例4-7 Employee.java

```
package com.horstmann.corejava;

// the classes in this file are part of this package

import java.util.*;

// import statements come after the package statement

/**
 * @version 1.10 1999-12-18
 * @author Cay Horstmann
 */
public class Employee
{
   public Employee(String n, double s, int year, int month, int day)
   {
      name = n;
      salary = s;
      GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
      // GregorianCalendar uses 0 for January
      hireDay = calendar.getTime();
   }

   public String getName()
   {
      return name;
   }

   public double getSalary()
   {
      return salary;
   }

   public Date getHireDay()
   {
```

```
      return hireDay;
   }

   public void raiseSalary(double byPercent)
   {
      double raise = salary * byPercent / 100;
      salary += raise;
   }

   private String name;
   private double salary;
   private Date hireDay;
}
```

4.7.4 包作用域

前面已经接触过访问修饰符public和private。标记为public的部分可以被任意的类使用；标记为private 的部分只能被定义它们的类使用。如果没有指定public或private，这个部分（类、方法或变量）可以被同一个包中的所有方法访问。

下面再仔细地看一下例4-2中的程序。在这个程序中，没有将Employee类定义为公有类，因此只有在同一个包（在此是默认包）中的其他类可以访问，例如EmployeeTest。对于类来说，这种默认是合乎情理的。但是，对于变量来说就有些不适宜了，因此变量必须显式地标记为private，不然的话将默认为包可见。显然，这样做会破坏封装性。问题主要出于人们经常忘记键入关键字private。在java.awt包中的Window类就是一个典型的示例。java.awt包是JDK提供的部分源代码：

```
public class Window extends Container
{
   String warningString;
   . . .
}
```

请注意，这里的warningString变量不是private！这意味着java.awt包中的所有类的方法都可以访问该变量，并将它设置为任意值（例如，"Trust me!"）。实际上，只有Window类的方法访问它，因此应该将它设置为私有变量。我们猜测可能是程序员匆忙之中忘记键入private修饰符了（为防止程序员内疚，我们没有说出他的名字，感兴趣的话，可以查看一下源代码）。



注释：奇怪的是，这个问题至今还没有得到纠正，即使我们在这本书的8个版本中已经指出了这一点。很明显，类库的实现者并没有读这本书。不仅如此，这个类还增加了一些新域，其中大约一半也不是私有的。

这真的会成为一个问题吗？答案是：视具体情况而定。在默认情况下，包不是一个封闭的实体。也就是说，任何人都可以向包中添加更多的类。当然，有敌意或低水平的程序员很可能利用包的可见性添加一些具有修改变量功能的代码。例如，在Java程序设计语言的早期版本中，只需要将下列这条语句放在类文件的开头，就可以很容易地将其他类混入java.awt包中：

```
package java.awt;
```

然后，把结果类文件放置在类路径某处的java/awt子目录下，就可以访问java.awt包的内部了。使用这一手段，可以对警告框进行设置（如图4-9所示）。

从1.2版开始，JDK的实现者修改了类加载器，明确地禁止加载用户自定义的、包名以“java.”开始的类！当然，用户自定义的类无法从这种保护中受益。然而，可以通过包密封（package sealing）机制来解决将各种包混杂在一起的问题。如果将一个包密封起来，就不能再向这个包添加类了。在第10章中，将介绍制作包含密封包的JAR文件的方法。

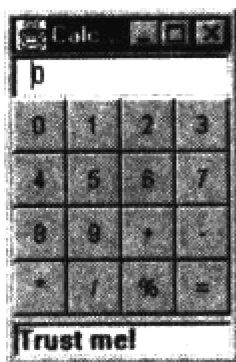


图4-9 修改在applet窗口中的警告字符串

4.8 类路径

在前面已经看到，类存储在文件系统的子目录中。类的路径必须与包名匹配。

另外，类文件也可以存储在JAR(Java归档)文件中。在一个JAR文件中，可以包含多个压缩形式的类文件和子目录，这样既可以节省又可以改善性能。在程序中用到第三方（third-party）的库文件时，通常会给出一个或多个需要包含的JAR文件。JDK也提供了许多的JAR文件，例如，在jre/lib子目录下包含数千个类库文件。有关创建JAR文件的详细内容将在第10章中讨论。

提示：JAR文件使用ZIP格式组织文件和子目录。可以使用所有ZIP实用程序查看内部的rt.jar以及其他的JAR文件。

为了使类能够被多个程序共享，需要做到下面几点：

1）把类放到一个目录中，例如/home/user/classdir。需要注意，这个目录是包树状结构的基目录。如果希望将com.horstmann.corejava.Employee类添加到其中，这个Employee.class类文件就必须位于子目录/home/user/classdir/com/horstmann/corejava中。

2）将JAR文件防在一个目录中，例如：/home/user/archives。

3）设置类路径（class path）。类路径是所有包含类文件的路径的集合。

在UNIX环境中，类路径中的不同项目之间采用冒号（:）分隔：

```
/home/user/classdir:.:/home/user/archives/archive.jar
```

而在Windows环境中，则以分号（;）分隔：

```
c:\classdir;.;c:\archives\archive.jar
```

在上述两种情况中，句点（.）表示当前目录。

类路径包括：

- 基目录 /home/user/classdir 或 c:\classes；
- 当前目录(.)；
- JAR文件/home/user/archives/archive.jar或c:\archives\archive.jar。

从Java SE 6开始，可以在JAR文件目录中指定通配符，如下：

```
/home/user/classdir:.:/home/user/archives/'*'
```

或者

```
c:\classdir;.;c:\archives\*
```

但在UNIX中，禁止使用* 以防止shell命令进一步扩展。

在归档目录中的所有JAR文件（但不包括.class文件）都包含在类路径中。

由于运行时库文件（rt.jar和在jre/lib与jre/lib/ext目录下的一些其他的JAR文件）会被自动地搜索，所以不必将它们显式地列在类路径中。

警告：javac编译器总是在当前的目录中查找文件，但Java虚拟机仅在类路径中有“.”目录的时候才查看当前目录。如果没有设置类路径，那也并不会产生什么问题，默认的类路径包含“.”目录。然而如果设置了类路径却忘记了包含“.”目录，则程序仍然可以通过编译，但不能运行。

类路径所列出的目录和归档文件是搜寻类的起始点。下面看一个类路径示例：

```
/home/user/classdir:.:/home/user/archives/archive.jar
```

假定虚拟机要搜寻com.horstmann.corejava.Employee类文件。它首先要查看存储在jre/lib和jre/lib/ext目录下的归档文件中所存放的系统类文件。显然，在那里找不到相应的类文件，然后再查看类路径。于是查看：

- /home/user/classdir/com/horstmann/corejava/Employee.class
- com/horstmann/corejava/Employee.class从当前目录开始
- com/horstmann/corejava/Employee.class inside /home/user/archives/archive.jar

编译器定位文件要比虚拟机复杂得多。如果引用了一个类，而没有指出这个类所在的包，那么编译器将首先查找包含这个类的包，并询查所有的import指令，确定其中是否包含了被引用的类。例如，假定源文件包含指令：

```
import java.util.*;
import com.horstmann.corejava.*;
```

并且源代码引用了Employee类。编译器将试图查找java.lang.Employee（因为java.lang包被默认导入）、java.util.Employee、com.horstmann.corejava.Employee和当前包中的Employee。对这个类路径的所有位置中所列出的每一个类进行逐一查看。如果找到了一个以上的类，就会产生编译错误（因为类必须是惟一的，而import语句的次序却无关紧要）。

编译器的任务不止这些，它还要查看源文件（Source files）是否比类文件新。如果是这样的话，那么源文件就会被自动地重新编译。在前面已经知道，仅可以导入其他包中的公有类。一个源文件只能包含一个公有类，并且文件名必须与公有类匹配。因此，编译器很容易定位公有类所在的源文件。当然，也可以从当前包中导入非公有类。这些类有可能定义在与类名不同的源文件中。如果从当前包中导入一个类，编译器就要搜索当前包中的所有源文件，以便确定哪个源文件定义了这个类。

设置类路径

最好采用-classpath（或-cp）选项指定类路径：

```
java -classpath /home/user/classdir:.:/home/user/archives/archive.jar MyProg.java
```

或者

```
java -classpath c:\classdir;.;c:\archives\archive.jar MyProg.java
```

所有的指令应该书写在一行中。将这样一个长的命令行放在一个shell脚本或一个批处理文件中是一个不错的主意。

利用-classpath选项设置类路径是首选的方法，也可以通过设置CLASSPATH环境变量完成这个操作。其详细情况依赖于所使用的shell。命令格式如下：

```
export CLASSPATH=/home/user/classdir:.:/home/user/archives/archive.jar
```

在C shell中，命令格式如下：

```
setenv CLASSPATH /home/user/classdir:.:/home/user/archives/archive.jar
```

在Windows shell，命令格式如下：

```
set CLASSPATH=c:\classdir;.;c:\archives\archive.jar
```

直到退出shell为止，类路径设置均有效。

警告：有人建议将CLASSPATH环境变量设置为永久不变的值。总的来说这是一个很糟糕的主意。人们有可能会忘记全局设置，因此，当使用的类没有正确地加载进来时，会感到很奇怪。一个应该受到谴责的示例是Windows中Apple的QuickTime安装程序。它进行了全局设置，CLASSPATH指向一个所需要的JAR文件，但并没有在类路径上包含当前路径。因此，当程序编译后却不能运行时，众多的Java程序员花费了很多精力去解决这个问题。

警告：有人建议绕开类路径，将所有的文件放在jre/lib/ext路径。这是一个极坏的主意，其原因主要有两个：当手工地加载其他的类文件时，如果将它们存放在扩展路径上，则不能正常地工作（有关类加载器的详细信息，请参看卷II第9章）。此外，程序员经常会忘记3个月前所存放文件的位置。当类加载器忽略了曾经仔细设计的类路径时，程序员会毫无头绪地在头文件中查找。事实上，加载的是扩展路径上已长时间遗忘的类。

4.9 文档注释

JDK包含一个很有用的工具，叫做javadoc，它可以由源文件生成一个HTML文档。事实上，在第3章讲述的联机API 文档就是通过对标准Java类库的源代码运行javadoc生成的。

如果在源代码中添加以专用的定界符/**开始的注释，那么可以很容易地生成一个看上去具有专业水准的文档。这是一种很好的方式，因为这种方式可以将代码与注释保存在一个地方。如果将文档存入一个独立的文件中，就有可能会随着时间的推移，出现代码和注释不一致的问题。然而，由于文档注释与源代码在同一个文件中，在修改源代码的同时，重新运行javadoc就可以轻而易举地保持两者的一致性。

4.9.1 注释的插入

javadoc实用程序（utility）从下面几个特性中抽取信息：

- 包
- 公有类与接口
- 公有的和受保护的方法

• 公有的和受保护的域

在第5章中将介绍受保护特性，在第6章将介绍接口。

应该为上面几部分编写注释。注释应该放置在所描述特性的前面。注释以 /** 开始，并以 */ 结束。

每个 /** . . . */ 文档注释在标记之后紧跟着*自由格式文本*（free-form text）。标记由 @ 开始，如@author或@param。

自由格式文本的*第一句*应该是一个*概要性的句子*。javadoc实用程序自动地将这些句子抽取出来形成概要页。

在自由格式文本中，可以使用HTML修饰符，例如，用于强调的 ...、用于设置等宽“打字机”字体的 <code>...</code>、用于着重强调的 ... 以及包含图像的<img ...>等。不过，一定不要使用<h1>或<hr>，因为它们会与文档的格式产生冲突。

注释：如果文档中有到其他文件的链接，例如，图像文件（用户界面的组件的图表或图像等），就应该将这些文件放到子目录doc-files中。javadoc实用程序将从源目录拷贝这些目录及其中的文件到文档目录中。在链接中需要使用doc-files目录，例如：<img src=“doc-files/uml.png” alt=“UML diagram”>。

4.9.2 类注释

类注释必须放在import语句之后，类定义之前。

下面是一个类注释的例子：

```
/**
 * A <code>Card</code> object represents a playing card, such
 * as "Queen of Hearts". A card has a suit (Diamond, Heart,
 * Spade or Club) and a value (1 = Ace, 2 . . . 10, 11 = Jack,
 * 12 = Queen, 13 = King)
 */
public class Card
{
   . . .
}
```

注释：没有必要在每一行的开始用星号*，例如：

```
/**
   A <code>Card</code> object represents a playing card, such
   as "Queen of Hearts". A card has a suit (Diamond, Heart,
   Spade or Club) and a value (1 = Ace, 2 . . . 10, 11 = Jack,
   12 = Queen, 13 = King).
*/
```

然而，大部分IDE提供了自动添加星号*，并且当注释行改变时，自动重新排列这些星号的功能。

4.9.3 方法注释

每一个方法注释必须放在所描述的方法之前。除了通用标记之外，还可以使用下面的标记：

- `@param variable description`

  这个标记将对当前方法的“param”（参数）部分添加一个条目。这个描述可以占据多行，并可以使用HTML标记。一个方法的所有 @param 标记必须放在一起。

- `@return description`

  这个标记将对当前方法添加“return”（返回）部分。这个描述可以跨越多行，并可以使用HTML标记。

- `@throws class description`

  这个标记将添加一个注释，用于表示这个方法有可能抛出异常。有关异常的详细内容将在第11章中讨论。

  下面是一个方法注释的示例：

```
/**
 * Raises the salary of an employee.
 * @param byPercent the percentage by which to raise the salary (e.g. 10 = 10%)
 * @return the amount of the raise
 */
public double raiseSalary(double byPercent)
{
   double raise = salary * byPercent / 100;
   salary += raise;
   return raise;
}
```

4.9.4 域注释

只需要对公有域（通常指的是静态常量）建立文档。例如，

```
/**
 * The "Hearts" card suit
 */
public static final int HEARTS = 1;
```

4.9.5 通用注释

下面的标记可以用在类文档的注释中。

- `@author name`

  这个标记将产生一个“author”（作者）条目。可以使用多个 @author 标记，每个 @author 标记对应一名作者。

- `@version text`

  这个标记将产生一个“version”（版本）条目。这里的text可以是对当前版本的任何描述。

下面的标记可以用于所有的文档注释。

- `@since text`

  这个标记将产生一个“since”（始于）条目。这里的text可以是对引入特性的版本描述。例如，@since version 1.7.1。

- `@deprecated text`

  这个标记将对类、方法或变量添加一个不再使用的注释。text中给出了取代的建议。例

如，

```
@deprecated Use <code>setVisible(true)</code> instead
```

通过@see和@link标记，可以使用超级链接，链接到javadoc文档的相关部分或外部文档。

• `@see reference`

这个标记将在“see also”部分增加一个超级链接。它可以用于类中，也可以用于方法中。这里的reference可以选择下列情形之一：

```
package.class#feature label
<a href="...">label</a>
"text"
```

第一种情况是最常见的。只要提供类、方法或变量的名字，javadoc就在文档中插入一个超链接。例如，

```
@see com.horstmann.corejava.Employee#raiseSalary(double)
```

建立一个链接到com.horstmann.corejava.Employee类的raiseSalary(double)方法的超链接。可以省略包名，甚至把包名和类名都省去，此时，链接将定位于当前包或当前类。

需要注意，一定要使用井号（#），而不要使用句号（.）分隔类名与方法名，或类名与变量名。Java编译器本身可以熟练地断定句点在分隔包、子包、类、内部类与方法和变量时的不同含义。但是javadoc实用程序就没有这么聪明了，因此必须对它提供帮助。

如果@see标记后面有一个<字符，就需要指定一个超链接。可以超链接到任何URL。例如：

```
@see <a href="www.horstmann.com/corejava.html">The Core Java home page</a>
```

在上述各种情况下，都可以指定一个可选的标签（label）作为锚链接（link anchor）。如果省略了label，用户看到的锚的名称就是目标代码名或URL。

如果@see标记后面有一个双引号（"）字符，文本就会显示在“see also”部分。例如，

```
@see "Core Java 2 volume 2"
```

可以为一个特性添加多个@see标记，但必须将它们放在一起。

• 如果愿意的话，还可以在注释中的任何位置放置指向其他类或方法的超级链接，以及插入一个专用的标记，例如，

```
{@link package.class#feature label}
```

这里的特性描述规则与@see标记规则一样。

4.9.6 包与概述注释

可以直接将类、方法和变量的注释放置在Java源文件中，只要用/** . . . */文档注释界定就可以了。但是，要想产生包注释，就需要在每一个包目录中添加一个单独的文件。可以有如下两个选择：

1）提供一个以package.html命名的HTML文件。在标记<BODY>...</BODY>之间的所有文本都会被抽取出来。

2）提供一个以package-info.java命名的Java文件。这个文件必须包含一个初始的以/**和*/界定的Javadoc注释，跟随在一个包语句之后。它不应该包含更多的代码或注释。

还可以为所有的源文件提供一个概述性的注释。这个注释将被放置在一个名为overview.html

的文件中，这个文件位于包含所有源文件的父目录中。标记<BODY>... </BODY>之间的所有文本将被抽取出来。当用户从导航栏中选择“Overview”时，就会显示出这些注释内容。

4.9.7 注释的抽取

这里，假设HTML文件将被存放在目录docDirectory下。执行以下步骤：

1）切换到包含想要生成文档的源文件目录。如果有嵌套的包要生成文档，例如com.horstmann.corejava，就必须切换到包含子目录com的目录（如果存在overview.html文件的话，这也是它的所在目录）。

2）如果是一个包，应该运行命令：

```
javadoc -d docDirectory nameOfPackage
```

或对于多个包生成文档，运行：

```
javadoc -d docDirectory nameOfPackage1 nameOfPackage2...
```

如果文件在默认包中，就应该运行：

```
javadoc -d docDirectory *.java
```

如果省略了-d docDirectory选项，那HTML文件就会被提取到当前目录下。这样有可能会带来混乱，因此不提倡这种做法。

可以使用多种形式的命令行选项对javadoc程序进行调整。例如，可以使用-author和-version选项在文档中包含@author和@version标记（默认情况下，这些标记会被省略）。另一个很有用的选项是-link，用来为标准类添加超链接。例如，如果使用命令

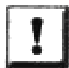
```
javadoc -link http://java.sun.com/javase/6/docs/api *.java
```

那么，所有的标准类库类都会自动地链接到Sun网站的文档。

如果使用-linksource选项，则每个源文件被转换为HTML（没有颜色编码，但包含行编号），并且每个类和方法名将转变为指向源代码的超链接。

有关其他的选项，请查阅javadoc实用程序的联机文档，http://java.sun.com/javase/javadoc。


注释：如果需要进一步的定制，例如，生成非HTML格式的文档，可以提供自定义的doclet，以便生成想要的任何输出形式。显然，这是一种特殊的需求，有关细节内容请查阅http://java.sun.com/j2se/javadoc的联机文档。


提示：DocCheck是一个很有用的doclet，在 http://java.sun.com/j2se/javadoc/doccheck/ 上。它可以为遗漏的文档注释搜索一组源程序文件。

4.10 类设计技巧

在结束本章之前，简单地介绍几点技巧。应用这些技巧可以使得设计出来的类更具有OOP的专业水准。

1）一定将数据设计为私有。

最重要的是：绝对不要破坏封装性。有时候，需要编写一个访问器方法或更改器方法，但是最好还是保持实例域的私有性。很多惨痛的经验告诉我们，数据的表示形式很可能会改变，

但它们的使用方式却不会经常发生变化。当数据保持私有时，它们的表示形式的变化不会对类的使用者产生影响，即使出现bug也易于检测。

2）一定要对数据初始化。

Java不对局部变量进行初始化，但是会对对象的实例域进行初始化。最好不要依赖于系统的默认值，而是应该显式地初始化所有的数据，具体的初始化方式可以是提供默认值，也可以是在所有构造器中设置默认值。

3）不要在类中使用过多的基本数据类型。

就是说，用其他的类代替多个相关的基本数据类型的使用。这样会使类更加易于理解且易于修改。例如，用一个称为Address的新的类替换下面的Customer类中的实例域：

```
private String street;
private String city;
private String state;
private int zip;
```

这样，可以很容易地顺应地址的变化，例如，需要增加对国际地址的处理。

4）不是所有的域都需要独立的域访问器和域更改器。

或许，需要获得或设置雇员的薪金。而一旦构造了雇员对象，就应该禁止更改雇用日期，并且在对象中，常常包含一些不希望别人获得或设置的实例域，例如，在Address类中，存放州缩写的数组。

5）使用标准格式进行类的定义。

一定采用下面的顺序书写类的内容：

公有访问特性部分

包作用域访问特性部分

私有访问特性部分

在每一部分中，应该按照下列顺序列出：

实例方法

静态方法

实例域

静态域

毕竟，类的使用者对公有接口要比对私有的实现细节更感兴趣，并且对方法要比对数据更感兴趣。

但是，哪一种风格更好并没有达成共识。Sun的程序设计风格建议 Java程序设计语言先书写域，后书写方法。无论采用哪种风格，重要的一点是要保持一致。

6）将职责过多的类进行分解。

这样说似乎有点含糊不清，究竟多少算是“过多”？每个人的看法不同。但是，如果明显地可以将一个复杂的类分解成两个更为简单的类，就应该将其分解（但另一方面，也不要走极端。设计10个类，每个类只有一个方法，显然也太小了）。

下面是一个反面的设计示例。

```
public class CardDeck // bad design
```

```
{
   public CardDeck() { . . . }
   public void shuffle() { . . . }
   public int getTopValue() { . . . }
   public int getTopSuit() { . . . }
   public void draw() { . . . }

   private int[] value;
   private int[] suit;
}
```

实际上，这个类实现了两个独立的概念：一副牌（含有shuffle方法和draw方法）和一张牌（含有查看面值和花色的方法）。另外，引入一个表示单张牌的Card类。现在有两个类，每个类完成自己的职责：

```
public class CardDeck
{
   public CardDeck() { . . . }
   public void shuffle() { . . . }
   public Card getTop() { . . . }
   public void draw() { . . . }

   private Card[] cards;
}

public class Card
{
   public Card(int aValue, int aSuit) { . . . }
   public int getValue() { . . . }
   public int getSuit() { . . . }

   private int value;
   private int suit;
}
```

7）*类名和方法名要能够体现它们的职责。*

与变量应该有一个能够反映其含义的名字一样，类也应该如此（在标准类库中，也存在着一些含义不明确的例子，如：Date类实际上是一个用于描述时间的类）。

命名类名的良好习惯是采用一个名词（Order）、前面有形容词修饰的名词（RushOrder）或动名词（有“-ing”后缀）修饰名词（例如，BillingAddress）。对于方法来说，习惯是访问器方法用小写get开头（getSalary），更改器方法用小写的set开头（setSalary）。

本章介绍了Java这种面向对象语言的有关对象和类的基础知识。为了真正做到面向对象，程序设计语言还必须支持继承和多态。Java提供了对这些特性的支持，具体内容将在下一章中介绍。

第5章 继 承

- ▲ 类、超类和子类
- ▲ Object：所有类的超类
- ▲ 泛型数组列表
- ▲ 对象包装器和自动打包
- ▲ 参数数量可变的方法
- ▲ 枚举类
- ▲ 反射
- ▲ 继承设计的技巧

第4章主要阐述了类和对象的概念，本章将学习面向对象程序设计的另外一个基本概念：继承（inheritance）。利用继承，人们可以基于已存在的类构造一个新类。继承已存在的类就是复用（继承）这些类的方法和域。在此基础上，还可以添加一些新的方法和域，以满足新的需求。这是Java程序设计中的一项核心技术。

与前一章相同，对于只使用过C、Visual Basic或COBOL这类面向过程的程序设计语言的读者来说，一定要仔细地阅读本章的内容。对于使用过C++或Smalltalk这类面向对象的程序设计语言的读者来说，对本章介绍的绝大部分内容可能已经比较熟悉。不过，Java与C++或其他面向对象的程序设计语言相比较，在实现继承的手段上存在着较大的差异。

另外，本章还阐述了反射（reflection）的概念。反射是指在程序运行期间发现更多的类及其属性的能力。这是一个功能强大的特性，使用起来也比较复杂。由于主要是开发软件工具的人员，而不是编写应用程序的人员对这项功能感兴趣，因此对于这部分内容，可以先浏览一下，待日后再返回来学习。

5.1 类、超类和子类

现在让我们重新回忆一下在前一章中讨论的Employee类。假设你在某个公司工作，这个公司中经理的待遇与普通雇员的待遇存在着一些差异。不过，他们之间也存在着很多相同的地方，例如，他们都领取薪水。只是普通雇员在完成本职任务之后仅领取薪水，而经理在完成了预期的业绩之后还能得到奖金。这种情形就需要使用继承。这是因为需要为经理定义一个新类Manager，以便增加一些新功能。但可以重用Employee类中已经编写的部分代码，并将其中的所有域保留下来。从理论上讲，在Manager与Employee之间存在着明显的“is-a”（是）关系，每个经理都是一名雇员：“is-a”关系是继承的一个明显特征。

下面是由继承Employee类来定义Manager类的格式，关键字extends表示继承。

```
class Manager extends Employee
{
   添加方法和域
}
```

C++注释：Java与C++定义继承类的方式十分相似。Java用关键字extends代替了C++中的冒号（:）。在Java中，所有的继承都是公有继承，而没有C++中的私有继承和保护继承。

关键字extends表明正在构造的新类派生于一个已存在的类。已存在的类被称为超类(superclass)、基类（base class）或父类（parent class）；新类被称为子类（subclass)、派生类(derived class）或孩子类（child class)。超类和子类是Java程序员最常用的两个术语，而其他语言种类的程序员可能更加偏爱使用父类和孩子类，这些都是继承时使用的术语。

尽管Employee类是一个超类，但并不是因为它位于子类之上或者拥有比子类更多的功能。恰恰相反，子类比超类拥有的功能更加丰富。例如，读过Manager类的源代码之后就会发现，Manager类比超类Employee封装了更多的数据，拥有更多的功能。

注释：前缀“超”和”子”来源于计算机科学和数学理论中的集合语言的术语。所有雇员组成的集合包含所有经理组成的集合。可以这样说，雇员集合是经理集合的超集，也可以说，经理集合是雇员集合的子集。

在Manager类中，增加了一个用于存储奖金信息的域，以及一个用于设置这个域的方法：

```
class Manager extends Employee
{
   . . .

   public void setBonus(double b)
   {
      bonus = b;
   }

   private double bonus;
}
```

这里定义的方法和域并没有什么特别之处。如果有一个Manager对象，就可以使用setBounds方法。

```
Manager boss = . . .;
boss.setBonus(5000);
```

当然，由于setBonus方法不是在Employee类中定义的，所以属于Employee类的对象不能使用它。

然而，尽管在Manager类中没有显式地定义getName和getHireDay等方法，但属于Manager类的对象却可以使用它们，这是因为Manager类自动地继承了超类Employee中的这些方法。

同样，从超类中还继承了name、salary和hireDay这3个域。这样一来，每个Manager类对象就包含了4个域：name、salary、hireDay和bonus。

在通过扩展超类定义子类的时候，仅需要指出子类与超类的不同之处。因此在设计类的时候，应该将通用的方法放在超类中，而将具有特殊用途的方法放在子类中，这种将通用的功能放到超类的做法，在面向对象程序设计中十分普遍。

然而，超类中的有些方法对子类Manager并不一定适用。例如，在Manager类中的getSalary方法应该返回薪水和奖金的总和。为此，需要提供一个新的方法来覆盖（override）超类中的这个方法：

```
class Manager extends Employee
{
```

```
    . . .
    public double getSalary()
    {
       . . .
    }
    . . .
}
```

应该如何实现这个方法呢？乍看起来似乎很简单，只要返回salary和bonus域的总和就可以了：

```
public double getSalary()
{
   return salary + bonus; // won't work
}
```

然而，这个方法并不能运行。这是因为Manager类的getSalary方法不能够直接地访问超类的私有域。也就是说，尽管每个Manager对象都拥有一个名为salary的域，但在Manager类的getSalary方法中并不能够直接地访问salary域。只有Employee类的方法才能够访问私有部分。如果Manager类的方法一定要访问私有域，就必须借助于公有的接口，Employee类中的公有方法getSalary正是这样一个接口。

现在，再试一下。将对salary域的访问替换成调用getSalary方法。

```
public double getSalary()
{
   double baseSalary = getSalary(); // still won't work
   return baseSalary + bonus;
}
```

上面这段代码仍然不能运行。问题出现在调用getSalary的语句上，这是因为Manager类也有一个getSalary方法（就是正在实现的这个方法），所以这条语句将会导致无限次地调用自己，直到整个程序崩溃为止。

这里需要指出：我们希望调用超类Employee中的getSalary方法，而不是当前类的这个方法。为此，可以使用特定的关键字super解决这个问题：

```
super.getSalary()
```

上述语句调用的是Employee类中的getSalary方法。下面是Manager类中getSalary方法的正确书写格式：

```
public double getSalary()
{
   double baseSalary = super.getSalary();
   return baseSalary + bonus;
}
```

注释：有些人认为super与this引用是类似的概念，实际上，这样比较并不太恰当。这是因为super不是一个对象的引用，不能将super赋给另一个对象变量，它只是一个指示编译器调用超类方法的特有关键字。

正像前面所看到的那样，在子类中可以增加域、增加方法或覆盖超类的方法，然而绝对不能删除继承的任何域和方法。

C++注释：在Java中使用关键字super调用超类的方法，而在C++中则采用超类名加上 :: 操作符的形式。例如，在Manager类的getSalary方法中，应该将super.getSalary替换为Employee::getSalary。

最后，看一下super在构造器中的应用。

```
public Manager(String n, double s, int year, int month, int day)
{
   super(n, s, year, month, day);
   bonus = 0;
}
```

这里的关键字super具有不同的含义。语句

```
super(n, s, year, month, day);
```

是"调用超类Employee中含有n、s、year、month和day参数的构造器"的简写形式。

由于Manager类的构造器不能访问Employee类的私有域，所以必须利用Employee类的构造器对这部分私有域进行初始化，我们可以通过super实现对超类构造器的调用。使用super调用构造器的语句必须是子类构造器的第一条语句。

如果子类的构造器没有显式地调用超类的构造器，则将自动地调用超类默认（没有参数）的构造器。如果超类没有不带参数的构造器，并且在子类的构造器中又没有显式地调用超类的其他构造器，则Java编译器将报告错误。

注释：回忆一下，关键字this有两个用途：一是引用隐式参数，二是调用该类其他的构造器。同样，super关键字也有两个用途：一是调用超类的方法，二是调用超类的构造器。在调用构造器的时候，这两个关键字的使用方式很相似。调用构造器的语句只能作为另一个构造器的第一条语句出现。构造参数既可以传递给本类（this）的其他构造器，也可以传递给超类（super）的构造器。

C++注释：在C++的构造函数中，使用初始化列表语法调用超类的构造函数，而不调用super。在C++中，Manager的构造函数如下所示：

```
Manager::Manager(String n, double s, int year, int month, int day) // C++
: Employee(n, s, year, month, day)
{
   bonus = 0;
}
```

重新定义Manager对象的getSalary方法之后，奖金就会自动地添加到经理的薪水中。

下面给出一个例子，其功能为创建一个新经理，并设置他的奖金：

```
Manager boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);
boss.setBonus(5000);
```

下面定义一个包含3个雇员的数组：

```
Employee[] staff = new Employee[3];
```

将经理和雇员都放到数组中：

```
staff[0] = boss;
staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
```

```
staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
```

输出每个人的薪水：

```
for (Employee e : staff)
   System.out.println(e.getName() + " " + e.getSalary());
```

运行这条循环语句将会输出下列结果：

```
Carl Cracker 85000.0
Harry Hacker 50000.0
Tommy Tester 40000.0
```

这里的staff[1]和staff[2]仅输出了基本薪水，这是因为它们对应的是Employee对象，而staff[0]对应的是Manager对象，它的getSalary方法将奖金与基本薪水加在了一起。

需要提到的是，e.getSalary( ) 能够确定应该执行哪个getSalary方法。请注意，尽管这里将e声明为Employee类型，但实际上e既可以引用Employee类型的对象，也可以引用Manager类型的对象。

当e引用Employee对象时，e.getSalary( )调用的是Employee类中的getSalary方法；当e引用Manager对象时，e.getSalary( )调用的是Manager类中的getSalary方法。虚拟机知道e实际引用的对象类型，因此能够正确地调用相应的类方法。

一个对象变量（例如，变量e）可以引用多种实际类型的现象被称为*多态*（polymorphism）。在运行时能够自动地选择调用哪个方法的现象称为*动态绑定*（dynamic binding）。在本章中将详细地讨论这两个概念。

C++注释：*在Java中，不需要将方法声明为虚拟方法。动态绑定是默认的处理方式。如果不希望让一个方法具有虚拟特征，可以将它标记为final（稍后将介绍关键字final）。*

例5-1的程序展示了Employee对象与Manager对象在薪水计算上的区别。

例5-1 ManagerTest.java

```
import java.util.*;

/**
 * This program demonstrates inheritance.
 * @version 1.21 2004-02-21
 * @author Cay Horstmann
 */
public class ManagerTest
{
   public static void main(String[] args)
   {
      // construct a Manager object
      Manager boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);
      boss.setBonus(5000);

      Employee[] staff = new Employee[3];

      // fill the staff array with Manager and Employee objects

      staff[0] = boss;
      staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
```

```
      staff[2] = new Employee("Tommy Tester", 40000, 1990, 3, 15);

      // print out information about all Employee objects
      for (Employee e : staff)
         System.out.println("name=" + e.getName() + ",salary=" + e.getSalary());
   }
}

class Employee
{
   public Employee(String n, double s, int year, int month, int day)
   {
      name = n;
      salary = s;
      GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
      hireDay = calendar.getTime();
   }

   public String getName()
   {
      return name;
   }

   public double getSalary()
   {
      return salary;
   }

   public Date getHireDay()
   {
      return hireDay;
   }

   public void raiseSalary(double byPercent)
   {
      double raise = salary * byPercent / 100;
      salary += raise;
   }

   private String name;
   private double salary;
   private Date hireDay;
}

class Manager extends Employee
{
   /**
    * @param n the employee's name
    * @param s the salary
    * @param year the hire year
    * @param month the hire month
    * @param day the hire day
    */
   public Manager(String n, double s, int year, int month, int day)
   {
      super(n, s, year, month, day);
      bonus = 0;
```

```
    }

    public double getSalary()
    {
       double baseSalary = super.getSalary();
       return baseSalary + bonus;
    }

    public void setBonus(double b)
    {
       bonus = b;
    }

    private double bonus;
}
```

5.1.1 继承层次

继承并不仅限于一个层次。例如，可以由Manager类派生Executive类。由一个公共超类派生出来的所有类的集合被称为*继承层次*（inheritance hierarchy），如图5-1所示。在继承层次中，从某个特定的类到其祖先的路径被称为该类的*继承链*（inheritance chain）。

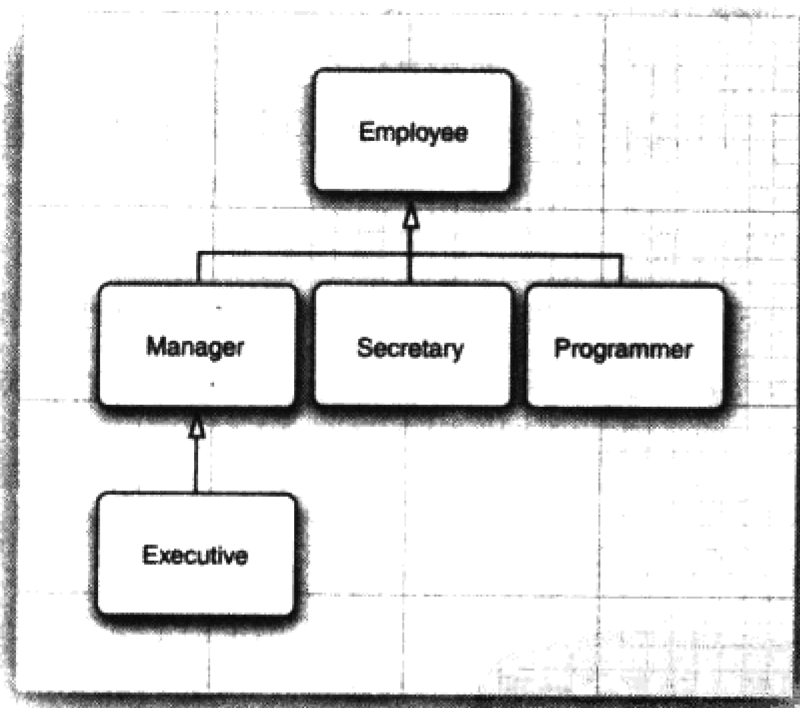


图5-1 Employee继承层次

通常，一个祖先类可以拥有多个子孙继承链。例如，可以由Employee类派生出子类Programmer或Secretary，它们与Manager类没有任何关系（有可能它们彼此之间也没有任何关系）。必要的话，可以将这个过程一直延续下去。


C++注释：*Java不支持多继承。有关Java中多继承功能的实现方式，请参看下一章有关接口的讨论。*

5.1.2 多态

有一个用来判断是否应该设计为继承关系的简单规则，这就是“is-a”规则，它表明子类

的每个对象也是超类的对象。例如，每个经理都是雇员，因此，将Manager类设计为Employee类的子类是显而易见的，反之不然，并不是每一名雇员都是经理。

“is-a”规则的另一种表述法是*置换法则*。它表明程序中出现超类对象的任何地方都可以用子类对象置换。例如，可以将一个子类的对象赋给超类变量。

```
Employee e;
e = new Employee(. . .);  // Employee object expected
e = new Manager(. . .); // OK, Manager can be used as well
```

在Java程序设计语言中，对象变量是*多态的*。一个Employee变量既可以引用一个Employee类对象，也可以引用一个Employee类的任何一个子类的对象（例如，Manager、Executive等等）。

从例5-1中，已经看到了置换法则的优点：

```
Manager boss = new Manager(. . .);
Employee[] staff = new Employee[3];
staff[0] = boss;
```

在这个例子中，变量staff[0]与boss引用同一个对象。但编译器将staff[0]看成Employee对象。

这意味着，可以这样调用

```
boss.setBonus(5000); // OK
```

但不能这样调用

```
staff[0].setBonus(5000); // ERROR
```

这是因为staff[0]声明的类型是Employee，而setBonus不是Employee类的方法。

然而，不能将一个超类的引用赋给子类变量。例如，下面的赋值是非法的

```
Manager m = staff[i]; // ERROR
```

原因很清楚：不是所有的雇员都是经理。如果赋值成功，m有可能引用了一个不是经理的Employee对象，当在后面调用m.setBonus(...)时就有可能发生运行时错误。

警告：在Java中，子类数组的引用可以转换成超类数组的引用，而不需要采用强制类型转换。例如，下面是一个经理数组

```
Manager[] managers = new Manager[10];
```

将它转换成Employee[]数组完全是合法的：

```
Employee[] staff = managers; // OK
```

这样做肯定不会有问题，请思考一下其中的缘由。毕竟，如果manager[i]是一个Manager，也一定是一个Employee。然而，实际上，将会发生一些令人惊讶的事情。要切记managers和staff引用的是同一个数组。现在看一下这条语句：

```
staff[0] = new Employee("Harry Hacker", ...);
```

编译器竟然接纳了这个赋值操作。但在这里，staff[0]与manager[0]引用的是同一个对象，似乎我们把一个普通雇员擅自归入经理行列中了。这是一种很忌讳发生的情形，当调用managers[0].setBonus(1000)的时候，将会导致调用一个不存在的实例域，进而搅乱相邻存储空间的内容。

为了确保不发生这类错误，所有数组都要牢记创建它们的元素类型，并负责监督仅将类型兼容的引用存储到数组中。例如，使用new managers[10] 创建的数组是一个经理数组。

如果试图存储一个Employee类型的引用就会引发ArrayStoreException异常。

5.1.3 动态绑定

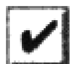
弄清调用对象方法的执行过程十分重要。下面是调用过程的详细描述：

1）编译器查看对象的声明类型和方法名。假设调用x.f(param)，且隐式参数x声明为C类的对象。需要注意的是：有可能存在多个名字为f，但参数类型不一样的方法。例如，可能存在方法f(int)和方法f(String)。编译器将会一一列举所有C类中名为f的方法和其超类中访问属性为public且名为f的方法。

至此，编译器已获得所有可能被调用的候选方法。

2）接下来，编译器将查看调用方法时提供的参数类型。如果在所有名为f的方法中存在一个与提供的参数类型完全匹配，就选择这个方法。这个过程被称为*重载解析*（overloading resolution）。例如，对于调用x.f("Hello")来说，编译器将会挑选f(String)，而不是f(int)。由于允许类型转换（int可以转换成double，Manager可以转换成Employee，等等），所以这个过程可能很复杂。如果编译器没有找到与参数类型匹配的方法，或者发现经过类型转换后有多个方法与之匹配，就会报告一个错误。

至此，编译器已获得需要调用的方法名字和参数类型。

 **注释**：前面曾经说过，方法的名字和参数列表称为方法的签名。例如，f(int)和f(String)是两个具有相同名字，不同签名的方法。如果在子类中定义了一个与超类签名相同的方法，那么子类中的这个方法就覆盖了超类中的这个相同签名的方法。

不过，返回类型不是签名的一部分，因此，在覆盖方法时，一定要保证返回类型的兼容性。在Java SE 5.0以前的版本中，要求返回类型必须是一样的。现在允许子类将覆盖方法的返回类型定义为原返回类型的子类型。例如，假设Employee类有

```
public Employee getBuddy() { ... }
```

在后面的子类Manager中，可以按照如下所示的方式覆盖这个方法

```
public Manager getBuddy() { ... } // OK in Java SE 5.0
```

我们说，这两个getBuddy方法具有可协变的返回类型。

3）如果是private方法、static方法、final方法（有关final修饰符的含义将在下一节讲述）或者构造器，那么编译器将可以准确地知道应该调用哪个方法，我们将这种调用方式称为*静态绑定*（static binding）。与此对应的是，调用的方法依赖于隐式参数的实际类型，并且在运行时实现动态绑定。在我们列举的示例中，编译器采用动态绑定的方式生成一条调用f (String)的指令。

4）当程序运行，并且采用动态绑定调用方法时，虚拟机一定调用与x所引用对象的实际类型最合适的那个类的方法。假设x 的实际类型是D，它是C类的子类。如果D类定义了方法f(String)，就直接调用它；否则，将在D类的超类中寻找f(String)，以此类推。

每次调用方法都要进行搜索，时间开销相当大。因此，虚拟机预先为每个类创建了一个*方法表*（method table），其中列出了所有方法的签名和实际调用的方法。这样一来，在真正调用方法的时候，虚拟机仅查找这个表就行了。在前面的例子中，虚拟机搜索D类的方法表，以便寻找与调用f(Sting) 相匹配的方法。这个方法既有可能是D.f(String)，也有可能是X.f(String)，

这里的X是D的超类。这里需要提醒一点，如果调用super.f(param)，编译器将对隐式参数超类的方法表进行搜索。

现在，查看一下例5-1中调用e.getSalary()的详细过程。e声明为Employee类型。Employee类只有一个名叫getSalary的方法，这个方法没有参数。因此，在这里不必担心重载解析的问题。

由于getSalary不是private方法、static方法或final方法，所以将采用动态绑定。虚拟机为Employee和Manager两个类生成方法表。在Employee的方法表中，列出了这个类定义的所有方法：

```
Employee:
   getName() -> Employee.getName()
   getSalary() -> Employee.getSalary()
   getHireDay() -> Employee.getHireDay()
   raiseSalary(double) -> Employee.raiseSalary(double)
```

实际上，上面列出的方法并不完整，稍后会看到Employee类有一个超类Object，Employee类从这个超类中还继承了许多方法，在此，我们略去了这些方法。

Manager方法表稍微有些不同。其中有三个方法是继承而来的，一个方法是重新定义的，还有一个方法是新增加的。

```
Manager:
   getName() -> Employee.getName()
   getSalary() -> Manager.getSalary()
   getHireDay() -> Employee.getHireDay()
   raiseSalary(double) -> Employee.raiseSalary(double)
   setBonus(double) -> Manager.setBonus(double)
```

在运行的时候，调用e.getSalary()的解析过程为：

1）首先，虚拟机提取e的实际类型的方法表。既可能是Employee、Manager的方法表，也可能是Employee类的其他子类的方法表。

2）接下来，虚拟机搜索定义getSalary签名的类。此时，虚拟机已经知道应该调用哪个方法。

3）最后，虚拟机调用方法。

动态绑定有一个非常重要的特性：无需对现存的代码进行修改，就可以对程序进行扩展。假设增加一个新类Executive，并且变量e有可能引用这个类的对象，我们不需要对包含调用e.getSalary() 的代码进行重新编译。如果e恰好引用一个Executive类的对象，就会自动地调用Executive.getSalary()方法。

警告：*在覆盖一个方法的时候，子类方法不能低于超类方法的可见性。特别是，如果超类方法是public，子类方法一定要声明为public。经常会发生这类错误：在声明子类方法的时候，遗漏了public修饰符。此时，编译器将会把它解释为试图降低访问权限。*

5.1.4 阻止继承：final类和方法

有时候，可能希望阻止人们利用某个类定义子类。不允许扩展的类被称为final类。如果在定义类的时候使用了final修饰符就表明这个类是final类。例如，假设希望阻止人们定义Executive类的子类，就可以在定义这个类的时候，使用final修饰符声明。声明格式如下所示：

```
final class Executive extends Manager
{
```

```
    . . .
}
```

类中的方法也可以被声明为final。如果这样做，子类就不能覆盖这个方法（final类中的所有方法自动地成为final方法）。例如

```
class Employee
{
   . . .
   public final String getName()
   {
      return name;
   }
   . . .
}
```

注释：前面曾经说过，域也可以被声明为final。对于final域来说，构造对象之后就不允许改变它们的值了。不过，如果将一个类声明为final，只有其中的方法自动地成为final，而不包括域。

将方法或类声明为final主要鉴于以下原因：确保它们不会在子类中改变语义。例如，Calendar类中的getTime和setTime方法都声明为final。这表明Calendar类的设计者负责实现Date类与日历状态之间的转换，而不允许子类处理这些问题。同样地，String类也是final类，这意味着不允许任何人定义String的子类。换言之，如果有一个String的引用，它引用的一定是一个String对象，而不可能是其他类的对象。

有些程序员认为：除非有足够的理由使用多态性，应该将所有的方法都声明为final。事实上，在C++和C#中，如果没有特别地说明，所有的方法都不具有多态性。这两种做法可能都有些偏激。我们提倡在设计类层次时，仔细地思考应该将哪些方法和类声明为final。

在早期的Java中，有些程序员为了避免动态绑定带来的系统开销而使用final关键字。如果一个方法没有被覆盖并且很短，编译器就能够对它进行优化处理，这个过程为称为内联(inlining)。例如，内联调用e.getName( )将被替换为访问e.name域。这是一项很有意义的改进，这是由于CPU在处理调用方法的指令时，使用的分支转移会扰乱预取指令的策略，所以，这被视为不受欢迎的。然而，如果getName在另外一个类中被覆盖，那么编译器就无法知道覆盖的代码将会做什么操作，因此也就不能对它进行内联处理了。

幸运的是，虚拟机中的即时编译器比传统编译器的处理能力强得多。这种编译器可以准确地知道类之间的继承关系，并能够检测出类中是否真正地存在覆盖给定的方法。如果方法很简短、被频繁调用且没有真正地被覆盖，那么即时编译器就会将这个方法进行内联处理。如果虚拟机加载了另外一个子类，而在这个子类中包含了对内联方法的覆盖，那么将会发生什么情况呢？优化器将取消对覆盖方法的内联。这个过程很慢，但却很少发生。

C++注释：C++中的方法在默认情况下不是动态绑定的，而是利用inline标记将方法调用替换成方法的源代码。不过，在C++中，没有提供任何机制阻止一个子类覆盖超类的方法。在C++中，也能够定义一个不允许被派生的类，但这需要使用一定的技巧，并且这样做也没有什么意义（这个问题将作为练习留给读者。提示：使用虚基类）。

5.1.5 强制类型转换

第3章曾经讲过，将一个类型强制转换成另外一个类型的过程被称为类型转换。Java程序设计语言提供了一种专门用于进行类型转换的表示法。例如：

```
double x = 3.405;
int nx = (int) x;
```

将表达式x的值转换成整数类型，舍弃了小数部分。

正像有时候需要将浮点型数值转换成整型数值一样，有时候也可能需要将某个类的对象引用转换成另外一个类的对象引用。对象引用的转换语法与数值表达式的类型转换类似，仅需要用一对圆括号将目标类名括起来，并放置在需要转换的对象引用之前就可以了。例如：

```
Manager boss = (Manager) staff[0];
```

进行类型转换的惟一原因是：在暂时忽视对象的实际类型之后，使用对象的全部功能。例如，在managerTest类中，由于某些项是普通雇员，所以staff数组必须是Employee对象的数组。我们需要将数组中引用经理的元素复原成Manager类，以便能够访问新增加的所有变量（需要注意，在前面的示例代码中，为了避免类型转换，我们做了一些特别的处理，即将boss变量存入数组之前，先用Manager对象对它进行初始化。而为了设置经理的奖金，必须使用正确的类型）。

大家知道，在Java中，每个对象变量都属于一个类型。类型描述了这个变量所引用的以及能够引用的对象类型。例如，staff[i]引用一个Employee对象（因此它还可以引用Manager对象）。

将一个值存入变量时，编译器将检查是否允许该操作。将一个子类的引用赋给一个超类变量，编译器是允许的。但将一个超类的引用赋给一个子类变量，必须进行类型转换，这样才能够通过运行时的检查。

如果试图在继承链上进行向下的类型转换，并且“谎报”有关对象包含的内容，会发生什么情况呢？

```
Manager boss = (Manager) staff[1]; // ERROR
```

运行这个程序时，Java运行时系统将报告这个错误，并产生一个ClassCastException异常。如果没有捕获这个异常，那么程序就会终止。因此，应该养成这样一个良好的程序设计习惯：在进行类型转换之前，先查看一下是否能够成功地转换。这个过程简单地使用instanceof运算符就可以实现。例如：

```
if (staff[1] instanceof Manager)
{
   boss = (Manager) staff[1];
   . . .
}
```

最后，如果这个类型转换不可能成功，编译器就不会进行这个转换。例如，下面这个类型转换：

```
Date c = (Date) staff[1];
```

将会产生编译性错误，这是因为Date不是Employee的子类。

综上所述：

- 只能在继承层次内进行类型转换。

• 在将超类转换成子类之前，应该使用instanceof进行检查。



注释：如果x为null，进行下列测试

```
x instanceof C
```

不会产生异常，只是返回false。之所以这样处理是因为null没有引用任何对象，当然也不会引用C类型的对象。

实际上，通过类型转换调整对象的类型并不是一种好的做法。在我们列举的示例中，大多数情况并不需要将Employee对象转换成Manager对象，两个类的对象都能够正确地调用getSalary方法，这是因为实现多态性的动态绑定机制能够自动地找到相应的方法。

只有在使用Manager中特有的方法时才需要进行类型转换，例如，setBonus方法。如果鉴于某种原因，发现需要通过Employee对象调用setBonus方法，那么就应该检查一下超类的设计是否合理。重新设计一下超类，并添加setBonus方法才是正确的选择。请记住，只要没有捕获ClassCastException异常，程序就会终止执行。在一般情况下，应该尽量少用类型转换和instanceof运算符。



C++注释：Java使用的类型转换语法来源于C语言“槽糕的旧时期”，但处理过程却有些像C++的dynamic_cast操作。例如，

```
Manager boss = (Manager) staff[1]; // Java
```

等价于

```
Manager* boss = dynamic_cast<Manager*>(staff[1]); // C++
```

它们之间只有一点重要的区别：当类型转换失败时，Java不会生成一个null对象，而是抛出一个异常。从这个意义上讲，有点像C++中的引用（reference）转换。真是令人生厌。在C++中，可以在一个操作中完成类型测试和类型转换。

```
Manager* boss = dynamic_cast<Manager*>(staff[1]); // C++
if (boss != NULL) . . .
```

而在Java中，需要将instanceof运算符和类型转换组合起来使用：

```
if (staff[1] instanceof Manager)
{
   Manager boss = (Manager) staff[1];
   . . .
}
```

5.1.6 抽象类

如果自下而上仰视类的继承层次结构，位于上层的类更具有通用性，甚至可能更加抽象。从某种角度看，祖先类更加通用，人们只将它作为派生其他类的基类，而不作为想使用的特定的实例类。例如，考虑一下对Employee类层次的扩展。一名雇员是一个人，一名学生也是一个人。下面将类Person和类Student添加到类的层次结构中。图5-2是这三个类之间的关系层次图。

为什么要花费精力进行这样高层次的抽象呢？每个人都有一些诸如姓名这样的属性。学生与雇员都有姓名属性，因此可以将getName方法放置在位于继承关系较高层次的通用超类中。

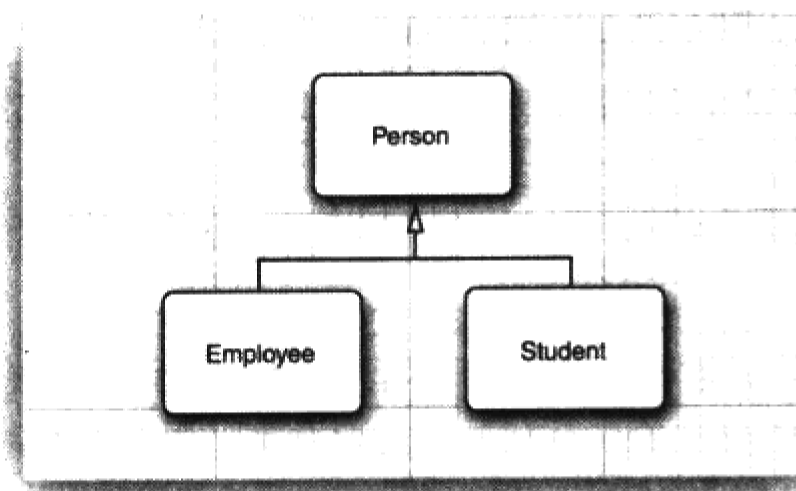


图5-2 Person与子类的关系层次示意图

现在，再增加一个getDescription方法，它可以返回对一个人的简短描述。例如：

```
an employee with a salary of $50,000.00
a student majoring in computer science
```

在Employee类和Student类中实现这个方法很容易。但是在Person类中应该提供什么内容呢？除了姓名之外，Person类一无所知。当然，可以让Person.getDescription( )返回一个空字符串。然而，还有一个更好的方法，就是使用abstract关键字，这样就完全不需要实现这个方法了。

```
public abstract String getDescription();
   // no implementation required
```

为了提高程序的清晰度，包含一个或多个抽象方法的类本身必须被声明为抽象的。

```
abstract class Person
{  . . .
   public abstract String getDescription();
}
```

除了抽象方法之外，抽象类还可以包含具体数据和具体方法。例如，Person类还保存着姓名和一个返回姓名的具体方法。

```
abstract class Person
{
   public Person(String n)
   {
      name = n;
   }

   public abstract String getDescription();

   public String getName()
   {
      return name;
   }

   private String name;
}
```

提示：许多程序员认为，在抽象类中不能包含具体方法。建议尽量将通用的域和方法（不管是否是抽象的）放在超类（不管是否是抽象的）中。

抽象方法充当着占位的角色，它们的具体实现在子类中。扩展抽象类可以有两种选择。一种是在子类中定义部分抽象方法或抽象方法也不定义，这样就必须将子类也标记为抽象类；另一种是定义全部的抽象方法，这样一来，子类就不是抽象的了。

例如，通过扩展抽象Person类，并实现getDescription方法来定义Student类。由于在Student类中不再含有抽象方法，所以不必将这个类声明为抽象的。

类即使不含抽象方法，也可以将类声明为抽象类。

抽象类不能被实例化。也就是说，如果将一个类声明为abstract，就不能创建这个类的对象。例如，表达式

```
new Person("Vince Vu")
```

是错误的，但可以创建一个具体子类的对象。

需要注意，可以定义一个抽象类的对象变量，但是它只能引用非抽象子类的对象。例如，

```
Person p = new Student("Vince Vu", "Economics");
```

这里的p是一个抽象类person的变量，它引用了一个非抽象子类Student的实例。



C++注释：在C++中，有一种在尾部用=0标记的抽象方法，被称为纯虚函数，例如：

```
class Person // C++
{
public:
   virtual string getDescription() = 0;
   . . .
};
```

只要有一个纯虚函数，这个类就是抽象类。在C++中，没有提供用于表示抽象类的特殊关键字。

下面通过抽象类Person扩展一个具体子类Student：

```
class Student extends Person
{
   public Student(String n, String m)
   {
      super(n);
      major = m;
   }

   public String getDescription()
   {
      return "a student majoring in " + major;
   }

   private String major;
}
```

在Student类中定义了getDescription方法。因此，在Student类中的全部方法都是非抽象的，这个类不再是抽象类。

在例5-2的程序中定义了抽象超类Person和两个具体子类Employee和Student。下面将雇员和学生对象赋给Person引用的数组。

```
Person[] people = new Person[2];
people[0] = new Employee(. . .);
people[1] = new Student(. . .);
```

然后，输出这些对象的姓名和信息描述：

```
for (Person p : people)
   System.out.println(p.getName() + ", " + p.getDescription());
```

有些人可能对下面这个调用感到困惑：

```
p.getDescription()
```

这不是调用了一个没有定义的方法吗？请牢记，由于不能构造抽象类Person的对象，所以变量p永远不会引用Person对象，而是引用诸如Employee或Student这样的具体子类对象，而在这些对象中都定义了getDescription方法。

是否可以省略Person超类中的抽象方法，而仅在Employee和Student子类中定义getDescription方法呢？如果这样的话，就不能通过变量p调用getDescription方法了。编译器只允许调用在类中声明的方法。

在Java程序设计语言中，抽象方法是一个重要的概念。在接口（interface）中将会看到更多的抽象方法。有关接口的详细介绍请参看第6章。

例5-2 PersonTest.java

```
import java.util.*;

/**
 * This program demonstrates abstract classes.
 * @version 1.01 2004-02-21
 * @author Cay Horstmann
 */
public class PersonTest
{
   public static void main(String[] args)
   {
      Person[] people = new Person[2];

      // fill the people array with Student and Employee objects
      people[0] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
      people[1] = new Student("Maria Morris", "computer science");

      // print out names and descriptions of all Person objects
      for (Person p : people)
         System.out.println(p.getName() + ", " + p.getDescription());
   }
}

abstract class Person
{
   public Person(String n)
   {
      name = n;
   }

   public abstract String getDescription();
```

```java

   public String getName()
   {
      return name;
   }

   private String name;
}

class Employee extends Person
{
   public Employee(String n, double s, int year, int month, int day)
   {
      super(n);
      salary = s;
      GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
      hireDay = calendar.getTime();
   }

   public double getSalary()
   {
      return salary;
   }

   public Date getHireDay()
   {
      return hireDay;
   }

   public String getDescription()
   {
      return String.format("an employee with a salary of $%.2f", salary);
   }

   public void raiseSalary(double byPercent)
   {
      double raise = salary * byPercent / 100;
      salary += raise;
   }

   private double salary;
   private Date hireDay;
}

class Student extends Person
{
   /**
    * @param n the student's name
    * @param m the student's major
    */
   public Student(String n, String m)
   {
      // pass n to superclass constructor
      super(n);
      major = m;
   }

```

```
    public String getDescription()
    {
       return "a student majoring in " + major;
    }

    private String major;
 }
```

5.1.7 受保护访问

大家都知道，最好将类中的域标记为private，而方法标记为public。任何声明为private的内容对其他类都是不可见的。前面已经看到，这对于子类来说也完全适用，即子类也不能访问超类的私有域。

然而，在有些时候，人们希望超类中的某些方法允许被子类访问，或允许子类的方法访问超类的某个域。为此，需要将这些方法或域声明为protected。例如，如果将超类Employee中的hireDay声明为proteced，而不是私有的，Manager中的方法就可以直接地访问它。

不过，Manager类中的方法只能够访问Manager对象中的hireDay域，而不能访问其他Employee对象中的这个域。这种限制有助于避免滥用受保护机制，使得子类只能获得访问受保护域的权利。

在实际应用中，要谨慎使用protected属性。假设需要将设计的类提供给其他程序员使用，而在这个类中设置了一些受保护域，由于其他程序员可以由这个类再派生出新类，并访问其中的受保护域。在这种情况下，如果需要对这个类的实现进行修改，就必须通知所有使用这个类的程序员。这违背了OOP提倡的数据封装原则。

受保护的方法更具有实际意义。如果需要限制某个方法的使用，就可以将它声明为protected。这表明子类（可能很熟悉祖先类）得到信任，可以正确地使用这个方法，而其他类则不行。

这种方法的一个最好的示例就是Object类中的clone方法，有关它的详细内容请参看第6章。



C++注释：事实上，Java中的受保护部分对所有子类及同一个包中的所有其他类都可见。这与C++中的保护机制稍有不同，Java中的protected概念要比C++中的安全性差。

下面归纳一下Java用于控制可见性的4个访问修饰符：

1）仅对本类可见——private。

2）对所有类可见——public。

3）对本包和所有子类可见——protected。

4）对本包可见——默认，所谓默认是指没有标明任何修饰符的情况，这是一种不太受欢迎的形式。

5.2 Object：所有类的超类

Object类是Java中所有类的最终祖先，在Java中每个类都是由它扩展而来的。但是并不需要这样写：

```
class Employee extends Object
```

如果没有明确地指出超类，Object就被认为是这个类的超类。由于在Java中，每个类都是由Object类扩展而来的，所以，熟悉这个类提供的所有服务十分重要。本章将介绍一些基本的内容，没有提到的部分请参看后面的章节或在线文档（在Object中有几个只在处理线程时才会被调用的方法，有关线程内容请参看卷Ⅱ）。

可以使用Object类型的变量引用任何类型的对象：

```
Object obj = new Employee("Harry Hacker", 35000);
```

当然，Object类型的变量只能用于作为各种值的通用持有者。要想对其中的内容进行具体的操作，还需要清楚对象的原始类型，并进行相应的类型转换：

```
Employee e = (Employee) obj;
```

在Java中，只有*基本类型*（primitive types）不是对象，例如，数值、字符和布尔类型的值都不是对象。所有的数组类型，不管是对象数组还是基本类型的数组都扩展于Object类。

```
Employee[] staff = new Employee[10];
obj = staff; // OK
obj = new int[10]; // OK
```

C++注释：*在C++中没有根类，不过，每个指针都可以转换成void*。*

5.2.1 Equals方法

Object类中的equals方法用于检测一个对象是否等于另外一个对象。在Object类中，这个方法将判断两个对象是否具有相同的引用。如果两个对象具有相同的引用，它们一定是相等的。从这点上看，将其作为默认操作也是合乎情理的。然而，对于多数类来说，这种判断并没有什么意义。例如，采用这种方式比较两个PrintStream对象是否相等就完全没有意义。然而，经常需要检测两个对象状态的相等性，如果两个对象的状态相等，就认为这两个对象是相等的。

例如，如果两个雇员对象的姓名、薪水和雇佣日期都一样，就认为它们是相等的（在实际的雇员数据库中，比较ID更有意义。利用下面这个示例演示equals方法的实现技巧）。

```
class Employee
{
   . . .
   public boolean equals(Object otherObject)
   {
      // a quick test to see if the objects are identical
      if (this == otherObject) return true;

      // must return false if the explicit parameter is null
      if (otherObject == null) return false;

      // if the classes don't match, they can't be equal
      if (getClass() != otherObject.getClass())
         return false;

      // now we know otherObject is a non-null Employee
      Employee other = (Employee) otherObject;
```

```
        // test whether the fields have identical values
        return name.equals(other.name)
           && salary == other.salary
           && hireDay.equals(other.hireDay);
     }
  }
```

getClass方法将返回一个对象所属的类，有关这个方法的详细内容稍后进行介绍。在检测中，只有在两个对象属于同一个类时，才有可能相等。

在子类中定义equals方法时，首先调用超类的equals。如果检测失败，对象就不可能相等。如果超类中的域都相等，就需要比较子类中的实例域。

```
class Manager extends Employee
{
   . . .
   public boolean equals(Object otherObject)
   {
      if (!super.equals(otherObject)) return false;
      // super.equals checked that this and otherObject belong to the same class
      Manager other = (Manager) otherObject;
      return bonus == other.bonus;
   }
}
```

5.2.2 相等测试与继承

如果隐式和显式的参数不属于同一个类，equals方法将如何处理呢？这是一个很有争议的问题。在前面的例子中，如果发现类不匹配，equals方法就返回false。但是，许多程序员却喜欢使用instanceof进行检测：

```
if (!(otherObject instanceof Employee)) return false;
```

这样做不但没有解决otherObject是子类的情况，并且还有可能会招致一些麻烦。这就是建议不要使用这种处理方式的原因所在。Java语言规范要求equals方法具有下面的特性：

1）**自反性**：对于任何非空引用x，x.equals(x) 应该返回true。

2）**对称性**：对于任何引用x和y，当且仅当y.equals(x)返回true，x.equals(y)也应该返回true。

3）**传递性**：对于任何引用x、y和z，如果x.equals(y)返回true，y.equals(z)返回true，x.equals(z) 也应该返回true。

4）**一致性**：如果x和y引用的对象没有发生变化，反复调用x.equals(y) 应该返回同样的结果。

5）对于任意非空引用x，x.equals(null) 应该返回false。

这些规则十分合乎情理，从而避免了类库实现者在数据结构中定位一个元素时还要考虑调用x.equals(y)，还是调用y.equals(x)的问题。

然而，就对称性来说，当参数不属于同一个类的时候需要仔细地思考一下。请看下面这个调用：

```
e.equals(m)
```

这里的e是一个Employee对象，m是一个Manager对象，并且两个对象具有相同的姓名、薪水和雇佣日期。如果在Employee.equals中用instanceof进行检测，则返回true。然而这意味着反过来

调用：

```
m.equals(e)
```

也需要返回true。对称性不允许这个方法调用返回false，或者抛出异常。

这就使得Manager类受到了束缚。这个类的equals方法必须能够用自己与任何一个Employee对象进行比较，而不考虑经理拥有的那部分特有信息！猛然间会让人感觉instanceof测试并不是完美无暇。

某些书的作者认为不应该利用getClass检测，因为这样不符合置换原则。有一个应用AbstractSet类的equals方法的典型例子，它将检测两个集合是否有相同的元素。AbstractSet类有两个具体子类：TreeSet和HashSet，它们分别使用不同的算法实现查找集合元素的操作。无论集合采用何种方式实现，都需要拥有对任意两个集合进行比较的功能。

然而，集合是相当特殊的一个例子，应该将AbstractSet.equals声明为final，这是因为没有任何一个子类需要重定义集合是否相等的语义（事实上，这个方法并没有被声明为final。这样做，可以让子类选择更加有效的算法对集合进行是否相等的检测）。

下面可以从两个截然不同的情况看一下这个问题：

- 如果子类能够拥有自己的相等概念，则对称性需求将强制采用getClass进行检测。
- 如果由超类决定相等的概念，那么就可以使用instanceof进行检测，这样可以在不同子类的对象之间进行相等的比较。

在雇员和经理的例子中，只要对应的域相等，就认为两个对象相等。如果两个Manager对象所对应的姓名、薪水和雇佣日期均相等，而奖金不相等，就认为它们是不相同的，因此，可以使用getClass检测。

但是，假设使用雇员的ID作为相等的检测标准，并且这个相等的概念适用于所有的子类，就可以使用instanceof进行检测，并应该将Employee.equals声明为final。

☑ **注释**：*在标准Java库中包含150多个equals方法的实现，包括使用instanceof检测、调用getClass检测、捕获ClassCastException或者什么也不做。*

下面给出编写一个完美的equals方法的建议：

1）显式参数命名为otherObject，稍后需要将它转换成另一个叫做other的变量。

2）检测this与otherObject是否引用同一个对象：

```
if (this == otherObject) return true;
```

这条语句只是一个优化。实际上，这是一种经常采用的形式。因为计算这个等式要比一个一个地比较类中的域所付出的代价小得多。

3）检测otherObject是否为null，如果为null，返回false。这项检测是很必要的。

```
if (otherObject == null) return false;
```

4）比较this与otherObject是否属于同一个类。如果equals的语义在每个子类中有所改变，就使用getClass检测：

```
if (getClass() != otherObject.getClass()) return false;
```

如果所有的子类都拥有统一的语义，就使用instanceof检测：

```
if (!(otherObject instanceof ClassName)) return false;
```

5）将otherObject转换为相应的类类型变量：

```
ClassName other = (ClassName) otherObject
```

6）现在开始对所有需要比较的域进行比较了。使用 == 比较基本类型域，使用equals比较对象域。如果所有的域都匹配，就返回true；否则返回false。

```
return field1 == other.field1
   && field2.equals(other.field2)
   && . . .;
```

如果在子类中重新定义equals，就要在其中包含调用super.equals(other)。

提示：对于数组类型的域，可以使用静态的Arrays.equals方法检测相应的数组元素是否相等。

警告：下面是实现equals方法的一种常见的错误。可以找到其中的问题吗？

```
public class Employee
{
   public boolean equals(Employee other)
   {
      return name.equals(other.name)
         && salary == other.salary
         && hireDay.equals(other.hireDay);
   }
   ...
}
```

这个方法声明的显式参数类型是Employee。其结果并没有覆盖Object类的equals方法，而是定义了一个完全无关的方法。

从Java SE 5.0开始，为了避免发生类型错误，可以使用@Override对覆盖超类的方法进行标记：

```
@Override public boolean equals(Object other)
```

如果出现了错误，并且正在定义一个新方法，编译器就会给出错误报告。例如，假设将下面的声明添加到Employee类中：

```
@Override public boolean equals(Employee other)
```

就会看到一个错误报告，这是因为这个方法并没有覆盖超类Object中的任何方法。

API java.util.Arrays 1.2

- `static Boolean equals(type[] a, type[] b) 5.0`

  如果两个数组长度相同，并且在对应的位置上数据元素也均相同，将返回true。数组的元素类型可以是Object,int,long,short,char,byte,boolean,float或double。

5.2.3 HashCode方法

散列码（hash code）是由对象导出的一个整型值。散列码是没有规律的。如果x和y是两个不同的对象，x.hashCode( )与y.hashCode( )基本上不会相同。在表5-1中列出了几个通过调用String类的hashCode方法得到的散列码。

String类使用下列算法计算散列码：

```
int hash = 0;
for (int i = 0; i < length(); i++)
   hash = 31 * hash + charAt(i);
```

由于hashCode方法定义在Object类中，因此每个对象都有一个默认的散列码，其值为对象的存储地址。看一下下面这个例子。

```
String s = "Ok";
StringBuilder sb = new StringBuilder(s);
System.out.println(s.hashCode() + " " + sb.hashCode());
String t = new String("Ok");
StringBuilder tb = new StringBuilder(t);
System.out.println(t.hashCode() + " " + tb.hashCode());
```

表5-2列出了结果。

表5-1 调用hashCode函数得到的散列码

String	Hash Code
Hello	69609650
Harry	69496448
Hacker	−2141031506

表5-2 String和String Buffers的散列码

Object	Hash Code
s	2556
sb	20526976
t	2556
tb	20527144

请注意，字符串s与t拥有相同的散列码，这是因为字符串的散列码是由内容导出的。而字符串缓冲sb与tb却有着不同的散列码，这是因为在StringBuffer类中没有定义hashCode方法，它的散列码是由Object类的默认hashCode方法导出的对象存储地址。

如果重新定义equals方法，就必须重新定义hashCode方法，以便用户可以将对象插入到散列表中（有关散列表的内容将在卷II的第2章中讨论）。

HashCode方法应该返回一个整型数值（也可以是负数），并合理地组合实例域的散列码，以便能够让各个不同的对象产生的散列码更加均匀。

例如，下面是Employee类的hashCode方法。

```
class Employee
{
   public int hashCode()
   {
      return 7 * name.hashCode()
         + 11 * new Double(salary).hashCode()
         + 13 * hireDay.hashCode();
   }
   . . .
}
```

Equals与hashCode的定义必须一致：如果x.equals(y)返回true，那么x.hashCode( )就必须与y.hashCode( )具有相同的值。例如，如果用定义的Employee.equals比较雇员的ID，那么hashCode方法就需要散列ID，而不是雇员的姓名或存储地址。

! **提示**：如果存在数组类型的域，那么可以使用静态的Arrays.hashCode方法计算一个散列码，这个散列码由数组元素的散列码组成。

java.lang.Object 1.0

- `int hashCode()`

  返回对象的散列码。散列码可以是任意的整数，包括正数或负数。两个相等的对象要求返回相等的散列码。

java.util.Arrays 1.2

- `static int hashCode(type[] a)  5.0`

  计算数组a的散列码。组成这个数组的元素类型可以是object, int, long, short, char, byte, boolean, float或double。

5.2.4 ToString方法

在Object中还有一个重要的方法，就是toString方法，它用于返回表示对象值的字符串。下面是一个典型的例子。Point类的toString方法将返回下面这样的字符串：

```
java.awt.Point[x=10,y=20]
```

绝大多数（但不是全部）的toString方法都遵循这样的格式：类的名字，随后是一对方括号括起来的域值。下面是Employee类中的toString方法的实现：

```
public String toString()
{
   return "Employee[name=" + name
      + ",salary=" + salary
      + ",hireDay=" + hireDay
      + "]";
}
```

实际上，还可以设计得更好一些。最好通过调用getClass( ).getName( ) 获得类名的字符串，而不要将类名硬加到toString方法中：

```
public String toString()
{
   return getClass().getName()
      + "[name=" + name
      + ",salary=" + salary
      + ",hireDay=" + hireDay
      + "]";
}
```

toString方法也可以供子类调用。

当然，设计子类的程序员也应该定义自己的toString方法，并将子类域的描述添加进去。如果超类使用了getClass( ).getName( )，那么子类只要调用super.toString( )就可以了。例如，下面是Manager类中的toString方法：

```
class Manager extends Employee
{
   . . .
   public String toString()
   {
      return super.toString()
```

```
            + "[bonus=" + bonus
            + "]";
    }
}
```

现在，Manager对象将打印输出如下所示的内容：

```
Manager[name=...,salary=...,hireDay=...][bonus=...]
```

随处可见toString方法的主要原因是：只要对象与一个字符串通过操作符“+”连接起来，Java编译就会自动地调用toString方法，以便获得这个对象的字符串描述。例如，

```
Point p = new Point(10, 20);
String message = "The current position is " + p;
   // automatically invokes p.toString()
```

提示：在调用x.toString( )的地方可以用""+x替代。这条语句将一个空串与x的字符串表示相连接。这里的x就是x.toString( )。与toString不同的是，如果x是基本类型，这条语句照样能够执行。

如果x是任意一个对象，并调用

```
System.out.println(x);
```

println方法就会直接地调用x.toString( )，并打印输出得到的字符串。

Object类定义了toString方法，用来打印输出对象所属的类名和散列码。例如，调用

```
System.out.println(System.out)
```

将输出下列内容：

```
java.io.PrintStream@2f6684
```

之所以得到这样的结果是因为PrintStream类的设计者没有覆盖toString方法。

警告：令人烦恼的是，数组继承了object类的toString方法，数组类型将按照旧的格式打印。例如：

```
int[] luckyNumbers = { 2, 3, 5, 7, 11, 13 };
String s = "" + luckyNumbers;
```

生成字符串“[I@1a46e30”（前缀[I表明是一个整型数组）。修正的方式是调用静态方法Arrays.toString。代码：

```
String s = Arrays.toString(luckyNumbers);
```

将生成字符串“[2,3,5,7,11,13]”。

要想打印多维数组（即，数组的数组）则需要调用Arrays.deepToString方法。

toString方法是一种非常有用的调试工具。在标准类库中，许多类都定义了toString方法，以便用户能够获得一些有关对象状态的必要信息。像下面这样显示调试信息非常有益：

```
System.out.println("Current position = " + position);
```

读者在第11章中将可以看到，更好的解决方法是：

```
Logger.global.info("Current position = " + position);
```

提示：强烈建议为自定义的每一个类增加toString方法。这样做不仅自己受益，而且所有使用这个类的程序员也会受益匪浅。

例5-3的程序实现了Employee类和Manager类的equals、hashcode和toString方法。

例5-3 EqualsTest.java

```
import java.util.*;

/**
 * This program demonstrates the equals method.
 * @version 1.11 2004-02-21
 * @author Cay Horstmann
 */
public class EqualsTest
{
   public static void main(String[] args)
   {
      Employee alice1 = new Employee("Alice Adams", 75000, 1987, 12, 15);
      Employee alice2 = alice1;
      Employee alice3 = new Employee("Alice Adams", 75000, 1987, 12, 15);
      Employee bob = new Employee("Bob Brandson", 50000, 1989, 10, 1);

      System.out.println("alice1 == alice2: " + (alice1 == alice2));

      System.out.println("alice1 == alice3: " + (alice1 == alice3));

      System.out.println("alice1.equals(alice3): " + alice1.equals(alice3));

      System.out.println("alice1.equals(bob): " + alice1.equals(bob));

      System.out.println("bob.toString(): " + bob);

      Manager carl = new Manager("Carl Cracker", 80000, 1987, 12, 15);
      Manager boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);
      boss.setBonus(5000);
      System.out.println("boss.toString(): " + boss);
      System.out.println("carl.equals(boss): " + carl.equals(boss));
      System.out.println("alice1.hashCode(): " + alice1.hashCode());
      System.out.println("alice3.hashCode(): " + alice3.hashCode());
      System.out.println("bob.hashCode(): " + bob.hashCode());
      System.out.println("carl.hashCode(): " + carl.hashCode());
   }
}

class Employee
{
   public Employee(String n, double s, int year, int month, int day)
   {
      name = n;
      salary = s;
      GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
      hireDay = calendar.getTime();
   }

   public String getName()
   {
      return name;
   }

```

```
   public double getSalary()
   {
      return salary;
   }

   public Date getHireDay()
   {
      return hireDay;
   }

   public void raiseSalary(double byPercent)
   {
      double raise = salary * byPercent / 100;
      salary += raise;
   }

   public boolean equals(Object otherObject)
   {
      // a quick test to see if the objects are identical
      if (this == otherObject) return true;

      // must return false if the explicit parameter is null
      if (otherObject == null) return false;

      // if the classes don't match, they can't be equal
      if (getClass() != otherObject.getClass()) return false;

      // now we know otherObject is a non-null Employee
      Employee other = (Employee) otherObject;

      // test whether the fields have identical values
      return name.equals(other.name) && salary == other.salary && hireDay.equals(other.hireDay);
   }

   public int hashCode()
   {
      return 7 * name.hashCode() + 11 * new Double(salary).hashCode() + 13 * hireDay.hashCode();
   }

   public String toString()
   {
      return getClass().getName() + "[name=" + name + ",salary=" + salary + ",hireDay=" + hireDay
            + "]";
   }

   private String name;
   private double salary;
   private Date hireDay;
}

class Manager extends Employee
{
   public Manager(String n, double s, int year, int month, int day)
   {
      super(n, s, year, month, day);
      bonus = 0;
   }
```

```
    public double getSalary()
    {
       double baseSalary = super.getSalary();
       return baseSalary + bonus;
    }


    public void setBonus(double b)
    {
       bonus = b;
    }

    public boolean equals(Object otherObject)
    {
       if (!super.equals(otherObject)) return false;
       Manager other = (Manager) otherObject;
       // super.equals checked that this and other belong to the same class
       return bonus == other.bonus;
    }

    public int hashCode()
    {
       return super.hashCode() + 17 * new Double(bonus).hashCode();
    }

    public String toString()
    {
       return super.toString() + "[bonus=" + bonus + "]";
    }

    private double bonus;
 }
```

API java.lang.Object 1.0

- `Class getClass()`

  返回包含对象信息的类对象。稍后会看到Java提供了类运行时的描述，它的内容被封装在Class类中。

- `boolean equals(Object otherObject)`

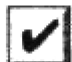
  比较两个对象是否相等，如果两个对象指向同一块存储区域，方法返回true；否则方法返回false。在自定义的类中，应该覆盖这个方法。

- `String toString()`

  返回描述该对象值的字符串。在自定义的类中，应该覆盖这个方法。

- `Object clone()`

  创建一个对象的副本。Java运行时系统将为新实例分配存储空间，并将当前的对象复制到这块存储区域中。



注释：复制对象是非常重要的，然而，却常常让人陷入一种相当棘手的境地，一不小心就落入陷阱。有关clone方法的详细内容将在第6章中阐述。

API java.lang.Class 1.0

- String getName()

  返回这个类的名字。
- Class getSuperclass()

  以Class对象的形式返回这个类的超类信息。

5.3 泛型数组列表

在许多程序设计语言中，特别是在C语言中，必须在编译时就确定整个数组的大小。程序员对此十分反感，因为这样做将迫使程序员做出一些不情愿的折中。例如，在一个部门中有多少雇员？肯定不会超过100人。一旦出现一个拥有150名雇员的大型部门呢？愿意为那些仅有10名雇员的部门浪费90名雇员占据的存储空间吗？

在Java中，情况就好多了。它允许在运行时确定数组的大小。

```
int actualSize = . . .;
Employee[] staff = new Employee[actualSize];
```

当然，这段代码并没有完全解决运行时动态更改数组的问题。一旦确定了数组的大小，改变它就不太容易了。在Java中，解决这个问题最简单的方法是使用Java中另外一个被称为ArrayList的类。它使用起来有点像数组，但在添加或删除元素时，具有自动调节数组容量的功能，而不需要为此编写任何代码。

在Java SE 5.0中，ArrayList是一个采用*类型参数*（type parameter）的*泛型类*（generic class）。为了指定数组列表保存的元素对象类型，需要用一对尖括号将类名括起来加在后面，例如，ArrayList <Employee>。在第13章中将可以看到如何自定义一个泛型类，这里并不需要了解任何技术细节就可以使用ArrayList类型。

下面声明和构造一个保存Employee对象的数组列表：

```
ArrayList<Employee> staff = new ArrayList<Employee>();
```

注释：Java SE 5.0以前的版本没有提供泛型类，而是有一个ArrayList类，其中保存类型为Object的元素，它是“自适应大小”的集合。如果一定要使用老版本的Java，则需要将所有的后缀<. . .>删掉。在Java SE 5.0以后的版本中，没有后缀<. . .>仍然可以使用ArrayList，它将被认为是一个删去了类型参数的“原始”类型。

注释：在Java程序设计语言的老版本中，程序员使用Vector类实现动态数组。不过，ArrayList类更加有效，没有任何理由一定要使用Vector类。

使用add方法可以将元素添加到数组列表中。例如，下面展示了如何将雇员对象添加到数组列表中的方法：

```
staff.add(new Employee("Harry Hacker", . . .));
staff.add(new Employee("Tony Tester", . . .));
```

数组列表管理着对象引用的一个内部数组。最终，数组的全部空间有可能被用尽。这就显现出数组列表的操作魅力：如果调用add且内部数组已经满了，数组列表就将自动地创建一个

更大的数组，并将所有的对象从较小的数组中拷贝到较大的数组中。

如果已经清楚或能够估计出数组可能存储的元素数量，就可以在填充数组之前调用ensureCapacity方法：

```
staff.ensureCapacity(100);
```

这个方法调用将分配一个包含100个对象的内部数组。然后调用100次add，而不用重新分配空间。

另外，还可以把初始容量传递给ArrayList构造器：

```
ArrayList<Employee> staff = new ArrayList<Employee>(100);
```

警告：分配数组列表，如下所示：

```
new ArrayList<Employee>(100) // capacity is 100
```

它与为新数组分配空间有所不同：

```
new Employee[100] // size is 100
```

数组列表的容量与数组的大小有一个非常重要的区别。如果为数组分配100个元素的存储空间，数组就有100个空位置可以使用。而容量为100个元素的数组列表只是拥有保存100个元素的潜力（实际上，重新分配空间的话，将会超过100），但是在最初，甚至完成初始化构造之后，数组列表根本就不含有任何元素。

size方法将返回数组列表中包含的实际元素数目。例如，

```
staff.size()
```

将返回staff数组列表的当前元素数量，它等价于数组a的a.length。

一旦能够确认数组列表的大小不再发生变化，就可以调用trimToSize方法。这个方法将存储区域的大小调整为当前元素数量所需要的存储空间数目。垃圾回收器将回收多余的存储空间。

一旦整理了数组列表的大小，添加新元素就需要花时间再次移动存储块，所以应该在确认不会添加任何元素时，再调用trimToSize。

C++注释：ArrayList类似于C++的vector模板。ArrayList与vector都是泛型类型。但是C++的vector模板为了便于访问元素重载了[ ]运算符。由于Java没有运算符重载，所以必须调用显式的方法。此外，C++向量是值拷贝。如果a和b是两个向量，赋值操作a = b将会构造一个与b长度相同的新向量a，并将所有的元素由b拷贝到a，而在Java中，这条赋值语句的操作结果是让a和b引用同一个数组列表。

API java.util.ArrayList<T> 1.2

- `ArrayList<T>()`

  构造一个空数组列表。

- `ArrayList<T>(int initialCapacity)`

  用指定容量构造一个空数组列表。

  参数：initalCapacity　　数组列表的最初容量

- `boolean add(T obj)`

  在数组列表的尾端添加一个元素。永远返回true。

参数：obj 添加的元素

- `int size()`

返回存储在数组列表中的当前元素数量。（这个值将小于或等于数组列表的容量。）

- `void ensureCapacity(int capacity)`

确保数组列表在不重新分配存储空间的情况下就能够保存给定数量的元素。

参数：capacity 需要的存储容量

- `void trimToSize()`

将数组列表的存储容量削减到当前尺寸。

5.3.1 访问数组列表元素

很遗憾，天下没有免费的午餐。数组列表自动扩展容量的便利增加了访问元素语法的复杂程度。其原因是ArrayList类并不是Java程序设计语言的一部分；它只是一个由某些人编写且被放在标准库中的一个实用类。

使用get和set方法实现访问或改变数组元素的操作，而不使用人们喜爱的 [ ] 语法格式。

例如，要设置第i个元素，可以使用：

```
staff.set(i, harry);
```

它等价于对数组a的元素赋值（数组的下标从0开始）：

```
a[i] = harry;
```

警告：只有i小于或等于数组列表的大小时，才能够调用list.set(i,x)。例如，下面这段代码是错误的：

```
ArrayList<Employee> list = new ArrayList<Employee>(100); // capacity 100, size 0
list.set(0, x); // no element 0 yet
```

使用add方法为数组添加新元素，而不要使用 set方法，它只能替换数组中已经存在的元素内容。

使用下列格式获得数组列表的元素：

```
Employee e = staff.get(i);
```

等价于：

```
Employee e = a[i];
```

注释：Java SE 5.0以前的版本没有泛型类，并且原始的ArrayList类提供的get方法毫无选择地返回Object，因此，get方法的调用者必须对返回值进行类型转换：

```
Employee e = (Employee) staff.get(i);
```

原始的ArrayList存在一定的危险性。它的add和set方法允许接受任意类型的对象。对于下面这个调用

```
staff.set(i, new Date());
```

编译不会给出任何警告，只有在检索对象并试图对它进行类型转换时，才会发现有问题。如果使用ArrayList<Employee>，编译器就会检测到这个错误。

下面这个技巧可以一举两得，既可以灵活地扩展数组，又可以方便地访问数组元素。首先，创建一个数组，并添加所有的元素。

```
ArrayList<X> list = new ArrayList<X>();
while (. . .)
{
   x = . . .;
   list.add(x);
}
```

执行完上述操作后，使用toArray方法将数组元素拷贝到一个数组中。

```
X[] a = new X[list.size()];
list.toArray(a);
```

除了在数组列表的尾部追加元素之外，还可以在数组列表的中间插入元素，使用带索引参数的add方法。

```
int n = staff.size() / 2;
staff.add(n, e);
```

为了插入一个新元素，位于n之后的所有元素都要向后移动一个位置。如果插入新元素后，数组列表的大小超过了容量，数组列表就会被重新分配存储空间。

同样地，可以从数组列表中删除一个元素。

```
Employee e = staff.remove(n);
```

位于这个位置之后的所有元素都向前移动一个位置，并且数组的大小减1。

对数组实施插入和删除元素的操作其效率比较低。对于小型数组来说，这一点不必担心。但如果数组存储的元素数比较多，又经常需要在中间位置插入、删除元素，就应该考虑使用链表了。有关链表操作的实现方式将在第13章中讲述。

在Java SE 5.0中，可以使用“for each”循环对数组列表遍历：

```
for (Employee e : staff)
   do something with e
```

这个循环和下列代码具有相同的效果

```
for (int i = 0; i < staff.size(); i++)
{
   Employee e = staff.get(i);
   do something with e
}
```

例5-4是对第4章中EmployeeTest做出修改后的程序。在这里，将Employee[ ]数组替换成了ArrayList<Employee>。请注意下面的变化：

- 不必指出数组的大小。
- 使用add将任意多的元素添加到数组中。
- 使用size()替代length计算元素的数目。
- 使用a.get(i)替代a[i]访问元素。

例5-4 ArrayListTest.java

```
import java.util.*;
```

```

/**
 * This program demonstrates the ArrayList class.
 * @version 1.1 2004-02-21
 * @author Cay Horstmann
 */
public class ArrayListTest
{
   public static void main(String[] args)
   {
      // fill the staff array list with three Employee objects
      ArrayList<Employee> staff = new ArrayList<Employee>();

      staff.add(new Employee("Carl Cracker", 75000, 1987, 12, 15));
      staff.add(new Employee("Harry Hacker", 50000, 1989, 10, 1));
      staff.add(new Employee("Tony Tester", 40000, 1990, 3, 15));

      // raise everyone's salary by 5%
      for (Employee e : staff)
         e.raiseSalary(5);

      // print out information about all Employee objects
      for (Employee e : staff)
         System.out.println("name=" + e.getName() + ",salary=" + e.getSalary() + ",hireDay="
               + e.getHireDay());
   }
}

class Employee
{
   public Employee(String n, double s, int year, int month, int day)
   {
      name = n;
      salary = s;
      GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
      hireDay = calendar.getTime();
   }

   public String getName()
   {
      return name;
   }

   public double getSalary()
   {
      return salary;
   }

   public Date getHireDay()
   {
      return hireDay;
   }

   public void raiseSalary(double byPercent)
   {
      double raise = salary * byPercent / 100;
      salary += raise;
```

```
    }

    private String name;
    private double salary;
    private Date hireDay;
}
```

API java.util.ArrayList<T> 1.2

- `void set(int index,T obj)`

  设置数组列表指定位置的元素值，这个操作将覆盖这个位置的原有内容。

  参数：index　　位置（必须介于0~size()－1之间）

  　　　obj　　新的值

- `T get(int index)`

  获得指定位置的元素值。

  参数：index　　获得的元素位置（必须介于0~size()－1之间）

- `void add(int index,T obj)`

  向后移动元素，以便插入元素。

  参数：index　　插入位置（必须介于0~size()－1之间）

  　　　obj　　新元素

- `T remove(int index)`

  删除一个元素，并将后面的元素向前移动。被删除的元素由返回值返回。

  参数：index　　被删除的元素位置（必须介于0~size()－1之间）

5.3.2 类型化与原始数组列表的兼容性

在采用Java SE 5.0以后的版本编写程序代码时，应该使用类型参数的数组列表，例如，ArrayList <Employee>。然而，有可能会发生与现存程序中的原始ArrayList类型进行交叉操作的情形。

假设有下面这个遗留下来的类：

```
public class EmployeeDB
{
   public void update(ArrayList list) { ... }
   public ArrayList find(String query) { ... }
}
```

可以将一个类型化的数组列表传递给update方法，而并不需要进行任何类型转换。

```
ArrayList<Employee> staff = ...;
employeeDB.update(staff);
```

也可以将staff对象传递给update方法。



警告：尽管编译器没有给出任何错误信息或警告，但是这样调用并不太安全。在update方法中，添加到数组列表中的元素可能不是Employee类型。在对这些元素进行检索时就会出现异常。听起来似乎很吓人，但思考一下就会发现，这与Java SE 5.0以前的版本是一

样的。虚拟机的完整性绝对没有受到威胁。在这种情形下，既没有降低安全性，也没有受益于编译时的检查。

相反地，将一个原始ArrayList赋给一个类型化ArrayList会得到一个警告。

```
ArrayList<Employee> result = employeeDB.find(query); // yields warning
```

✔ **注释**：为了能够看到警告性错误的文字信息，要将编译选项置为-Xlint:unchecked。

使用类型转换并不能避免出现警告。

```
ArrayList<Employee> result = (ArrayList<Employee>)
   employeeDB.find(query); // yields another warning
```

这样，将会得到另外一个警告信息，被告之类型转换有误。

这就是Java中不尽人意的参数化类型的限制所带来的结果。鉴于兼容性的考虑，编译器在对类型转换进行检查之后，如果没有发现违反规则的现象，就将所有的类型化数组列表转换成原始ArrayList对象。在程序运行时，所有的数组列表都是一样的，即没有虚拟机中的类型参数。因此，类型转换（ArrayList）和（ArrayList<Employee>）将执行相同的运行时检查。

在这种情形下，不必做什么。只要在与遗留的代码进行交叉操作时，研究一下编译器的警告性提示，并确保这些警告不会造成太严重的后果就行了。

5.4 对象包装器与自动打包

有时，需要将int这样的基本类型转换为对象。所有的基本类型都有一个与之对应的类。例如，Integer类对应基本类型int。通常，这些类称为*包装器*（wrapper）。这些对象包装器类拥有很鲜明的名字：Integer、Long、Float、Double、Short、Byte、Character、Void和Boolean（前6个类派生于公共的超类Number）。对象包装器类是不可变的，即一旦构造了包装器，就不允许更改包装在其中的值。同时，对象包装器类还是final，因此不能定义它们的子类。

假设想定义一个整型数组列表。而尖括号中的类型参数不允许是基本类型，也就是说，不允许写成ArrayList<int>。这里就用到了Integer对象包装器类。我们可以声明一个Integer对象的数组列表。

```
ArrayList<Integer> list = new ArrayList<Integer>();
```

✘ **警告**：由于每个值分别包装在对象中，所以ArrayList<Integer>的效率远远低于int[ ]数组。因此，应该用它构造小型集合，其原因是此时程序员操作的方便性要比执行效率更加重要。

Java SE 5.0的另一个改进之处是更加便于添加或获得数组元素。下面这个调用

```
list.add(3);
```

将自动地变换成

```
list.add(new Integer(3));
```

这种变换被称为自动打包（autoboxing）。

注释：大家可能认为自动打包（autowrapping）更加合适，而“装箱（boxing）”这个词源自于C#。

相反地，当将一个Integer对象赋给一个int值时，将会自动地拆包。也就是说，编译器将下列语句：

```
int n = list.get(i);
```

翻译成

```
int n = list.get(i).intValue();
```

甚至在算术表达式中也能够自动地打包和拆包。例如，可以将自增操作符应用于一个包装器引用：

```
Integer n = 3;
n++;
```

编译器将自动地插入一条拆开对象包的指令，然后进行自增计算，最后再将结果打入对象包内。

在很多情况下，容易有一种假象，即基本类型与它们的对象包装器是一样的，只是它们的相等性不同。大家知道，==运算符也可以应用于对象包装器对象，只不过检测的是对象是否指向同一个存储区域，因此，下面的比较通常不会成立：

```
Integer a = 1000;
Integer b = 1000;
if (a == b) ...
```

然而，Java实现却有可能（may）让它成立。如果将经常出现的值包装到同一个对象中，这种比较就有可能成立。这种不确定的结果并不是我们所希望的。解决这个问题的办法是在两个包装器对象比较时调用equals方法。

注释：自动打包规范要求boolean、byte、char≤127，介于－128～127之间的short和int被包装到固定的对象中。例如，如果在前面的例子中将a和b初始化为100，对它们进行比较的结果一定成立。

最后强调一下，打包和拆包是编译器认可的，而不是虚拟机。编译器在生成类的字节码时，插入必要的方法调用。虚拟机只是执行这些字节码。

使用数值对象包装器还有另外一个好处。Java设计者发现，可以将某些基本方法放置在包装器中，例如，将一个数字字符串转换成数值。

要想将字符串转换成整型，可以使用下面这条语句：

```
int x = Integer.parseInt(s);
```

这里与Integer对象没有任何关系，parseInt是一个静态方法。但Integer类是放置这个方法的一个好地方。

API注释说明了Integer类中包含的一些重要方法。其他数值类也实现了相应的方法。

警告：有些人认为包装器类可以用来实现修改数值参数的方法，然而这是错误的。在第4章中曾经讲到，由于Java方法都是值传递，所以不可能编写一个下面这样的能够增加整型参数值的Java方法。

```
public static void triple(int x) // won't work
{
   x = 3 * x; // modifies local variable
}
```

将int替换成Integer又会怎样呢？

```
public static void triple(Integer x) // won't work
{
   ...
}
```

问题是Integer对象是不可变的：包含在包装器中的内容不会改变。不能使用这些包装器类创建修改数值参数的方法。

如果想编写一个修改数值参数值的方法，就需要使用在org.omg.CORBA包中定义的持有者（holder）类型，包括IntHolder、BooleanHolder等等。每个持有者类型都包含一个公有（!）域值，通过它可以访问存储在其中的值。

```
public static void triple(IntHolder x)
{
   x.value = 3 * x.value;
}
```

API java.lang.Integer 1.0

- `int intValue()`

  以int的形式返回Integer对象的值（在Number类中覆盖了intValue方法）。
- `static String toString(int i)`

  以一个新String对象的形式返回给定数值i的十进制表示。
- `static String toString(int i,int radix)`

  返回数值i的基于给定radix参数进制的表示。
- `static int parseInt(String s)`
- `static int parseInt(String s,int radix)`

  返回字符串s表示的整型数值，给定字符串表示的是十进制的整数（第一种方法），或者是radix参数进制的整数（第二种方法）。
- `static Integer valueOf(String s)`
- `Static Integer value Of(String s, int  radix)`

  返回用s表示的整型数值进行初始化后的一个新Integer对象，给定字符串表示的是十进制的整数（第一种方法），或者是radix参数进制的整数（第二种方法）。

API java.text.NumberFormat 1.1

- `Number parse(String s)`

  返回数字值，假设给定的String表示了一个数值。

5.5 参数数量可变的方法

在Java SE 5.0以前的版本中，每个Java方法都有固定数量的参数。然而，现在的版本提供

了可以用可变的参数数量调用的方法（有时称为“可变参”方法）。

前面已经看到过这样的方法：printf。例如，下面的方法调用：

```
System.out.printf("%d", n);
```

和

```
System.out.printf("%d %s", n, "widgets");
```

在上面两条语句中，尽管一个调用包含两个参数，另一个调用包含三个参数，但它们调用的都是同一个方法。printf方法是这样定义的：

```
public class PrintStream
{
   public PrintStream printf(String fmt, Object... args) { return format(fmt, args); }
}
```

这里的省略号...是Java代码的一部分，它表明这个方法可以接收任意数量的对象（除fmt参数之外）。

实际上，printf方法接收两个参数，一个是格式字符串，另一个是Object[ ]数组，其中保存着所有的参数（如果调用者提供的是整型数组或者其他基本类型的值，自动打包功能将把它们转换成对象）。现在将扫描fmt字符串，并将第i个格式说明符与args[i]的值匹配起来。

换句话说，对于printf的实现者来说，Object...参数类型与Object[ ]完全一样。

编译器需要对printf的每次调用进行转换，以便将参数绑定到数组上，并在必要的时候进行自动打包：

```
System.out.printf("%d %s", new Object[] { new Integer(n), "widgets" } );
```

用户自己也可以定义可变参数的方法，并将参数指定为任意类型，甚至是基本类型。下面是一个简单的示例：其功能为计算若干个数值的最大值。

```
public static double max(double... values)
{
   double largest = Double.MIN_VALUE;
   for (double v : values) if (v > largest) largest = v;
   return largest;
}
```

可以像下面这样调用这个方法：

```
double m = max(3.1, 40.4, -5);
```

编译器将new double[ ] {3.1, 40.4, -5}传递给max方法。



注释：允许将一个数组传递给可变参数方法的最后一个参数。例如：

```
System.out.printf("%d %s", new Object[] { new Integer(1), "widgets" } );
```

因此，可以将已经存在且最后一个参数是数组的方法重新定义为可变参数的方法，而不会破坏任何已经存在的代码。例如，MessageFormat.format在Java SE 5.0就采用了这种方式。甚至可以将main方法声明为下列形式：

```
public static void main(String... args)
```

5.6 枚举类

读者在第3章已经看到，如何在Java SE 5.0以后的版本中定义枚举类型。下面是一个典型的例子：

```
public enum Size { SMALL, MEDIUM, LARGE, EXTRA_LARGE };
```

实际上，这个声明定义的类型是一个类，它刚好有4个实例，在此尽量不要构造新对象。

因此，在比较两个枚举类型的值时，永远不需要调用equals，而直接使用“==”就可以了。

如果需要的话，可以在枚举类型中添加一些构造器、方法和域。当然，构造器只是在构造枚举常量的时候被调用。下面是一个示例：

```
enum Size
{
   SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");

   private Size(String abbreviation) { this.abbreviation = abbreviation; }
   public String getAbbreviation() { return abbreviation; }

   private String abbreviation;
}
```

所有的枚举类型都是Enum类的子类。它们继承了这个类的许多方法。其中最有用的一个是toString，这个方法能够返回枚举常量名。例如，Size.SMALL.toString( )将返回字符串“SMALL”。

toString的逆方法是静态方法valueOf。例如，语句：

```
Size s = (Size) Enum.valueOf(Size.class, "SMALL");
```

将s设置成Size.SMALL。

每个枚举类型都有一个静态的values方法，它将返回一个包含全部枚举值的数组。例如，如下调用

```
Size[] values = Size.values();
```

返回包含元素Size.SMALL,Size.MEDIUM,Size.LARGE和Size.EXTRA_LARGE的数组。

ordinal方法返回enum声明中枚举常量的位置，位置从0开始计数。例如：Size.MEDIUM.ordinal()返回1。

例5-5程序演示了枚举类型的工作方式。

注释：如同Class类一样，鉴于简化的考虑，Enum类省略了一个类型参数。例如，实际上，应该将枚举类型Size扩展为Enum<Size>。类型参数在compareTo方法中使用（类型参数方法在第6章中介绍，类型参数在第12章中介绍）。

例5-5 EnumTest.java

```
import java.util.*;

/**
 * This program demonstrates enumerated types.
```

```
 * @version 1.0 2004-05-24
 * @author Cay Horstmann
 */
public class EnumTest
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);
      System.out.print("Enter a size: (SMALL, MEDIUM, LARGE, EXTRA_LARGE) ");
      String input = in.next().toUpperCase();
      Size size = Enum.valueOf(Size.class, input);
      System.out.println("size=" + size);
      System.out.println("abbreviation=" + size.getAbbreviation());
      if (size == Size.EXTRA_LARGE)
         System.out.println("Good job--you paid attention to the _.");
   }
}

enum Size
{
   SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");

   private Size(String abbreviation) { this.abbreviation = abbreviation; }
   public String getAbbreviation() { return abbreviation; }

   private String abbreviation;
}
```



java.lang.Enum <E> 5.0

- `static Enum valueOf(Class enumClass,String name)`

  返回指定名字、给定类的枚举常量。
- `String toString( )`

  返回枚举常量名。
- `int ordinal()`

  返回枚举常量在enum声明中的位置，位置从0开始计数。
- `int compareTo(E other)`

  如果枚举常量出现在other之前，则返回一个负值；如果this==other，则返回0；否则，返回正值。枚举常量的出现次序在enum声明中给出。

5.7 反射

反射库（reflection library）提供了一个非常丰富且精心设计的工具集，以便编写能够动态操纵Java代码的程序。这项功能被大量地应用于JavaBeans中，它是Java组件的体系结构（有关JavaBeans的详细内容在卷II中阐述）。使用反射，Java可以支持Visual Basic用户习惯使用的工具。特别是在设计或运行中添加新类时，能够快速地应用开发工具动态地查询新添加类的能力。

能够分析类能力的程序被称为*反射*（reflective）。反射机制的功能极其强大。在下面的章节中可以看到，可以用反射机制：

- 在运行中分析类的能力。
- 在运行中查看对象，例如，编写一个toString方法供所有类使用。
- 实现数组的操作代码。
- 利用Method对象，这个对象很像C++中的函数指针。

反射是一种功能强大且复杂的机制。使用它的主要对象是工具构造者，而不是应用程序员。如果仅对设计应用程序感兴趣，而对构造工具不感兴趣，可以跳过本章的剩余部分，稍后再返回来学习。

5.7.1 Class类

在程序运行期间，Java运行时系统始终为所有的对象维护一个被称为运行时的类型标识。这个信息保存着每个对象所属的类足迹。虚拟机利用运行时信息选择相应的方法执行。

然而，可以通过专门的Java类访问这些信息。保存这些信息的类被称为Class，这个名字很容易让人混淆。Object类中的getClass( )方法将会返回一个Class类型的实例。

```
Employee e;
. . .
Class cl = e.getClass();
```

如同用一个Employee对象表示一个特定的雇员属性一样，一个Class对象将表示一个特定类的属性。最常用的Class方法是getName。这个方法将返回类的名字。例如，下面这条语句：

```
System.out.println(e.getClass().getName() + " " + e.getName());
```

如果e是一个雇员，则会打印输出：

```
Employee Harry Hacker
```

如果e是经理，则会打印输出:

```
Manager Harry Hacker
```

如果类在一个包里，包的名字也作为类名的一部分：

```
Date d = new Date();
Class cl = d.getClass();
String name = cl.getName(); // name is set to "java.util.Date"
```

还可以调用静态方法forName获得类名对应的Class对象。

```
String className = "java.util.Date";
Class cl = Class.forName(className);
```

如果类名保存在字符串中，并可在运行中改变，就可以使用这个方法。当然，这个方法只有在className是类名或接口名时才能够执行。否则，forName方法将抛出一个checked exception（已检查异常）。无论何时使用这个方法，都应该提供一个异常处理器（exception handler）。如何提供一个异常处理器，请参看本节后面的“捕获异常”部分。

! 提示：*在启动时，包含main方法的类被加载。它会加载所有需要的类。这些被加载的类又要加载它们需要的类，以此类推。对于一个大型的应用程序来说，这将会消耗很多时间，用户会因此感到不耐烦。可以使用下面这个技巧给用户一种启动速度比较快的幻觉。不过，要确保包含main方法的类没有显式地引用其他的类。首先，显示一个启动画面；*

然后，通过调用Class.forName手工地加载其他的类。

获得Class类对象的第三种方法非常简单。如果T是任意的Java类型，T.class将代表匹配的类对象。例如：

```
Class cl1 = Date.class; // if you import java.util.*;
Class cl2 = int.class;
Class cl3 = Double[].class;
```

请注意，一个Class对象实际上表示的是一个类型，而这个类型未必一定是一种类。例如，int不是类，但int.class是一个Class类型的对象。

注释：从Java SE 5.0开始，Class类被参数化。例如，Class<Employee>的类型是Employee.class。没有说明这个问题的原因是：它将已经抽象的概念更加复杂化了。在大多数实际问题中，可以忽略类型参数，而使用原始的Class类。有关这个问题更详细的论述请参看第13章。

警告：鉴于历史原因，getName方法在应用于数组类型的时候会返回一个很奇怪的名字：
- `Double[ ].class.getName( )` 返回"`[Ljava.lang.Double;`"。
- `int[ ].class.getName( )` 返回"`[I`"。

虚拟机为每个类型管理一个Class对象。因此，可以利用 == 运算符实现两个类对象比较的操作。例如，

```
if (e.getClass() == Employee.class) . . .
```

还有一个很有用的方法newInstance( )，可以用来快速地创建一个类的实例。例如，

```
e.getClass().newInstance();
```

创建了一个与e具有相同类类型的实例。newInstance方法调用默认的构造器（没有参数的构造器）初始化新创建的对象。如果这个类没有默认的构造器，就会抛出一个异常。

将forName与newInstance配合起来使用，可以根据存储在字符串中的类名创建一个对象。

```
String s = "java.util.Date";
Object m = Class.forName(s).newInstance();
```

注释：如果需要以这种方式向希望按名称创建的类的构造器提供参数，就不要使用上面那条语句，而必须使用Constructor类中的newInstance方法。

C++注释：newInstance方法对应C++中虚拟构造器的习惯用法。然而，C++中的虚拟构造器不是一种语言特性，需要由专门的库支持。Class类与C++中的type_info类相似，getClass方法与C++中的tyieid运算符等价。但Java中的Class比C++中的type_info的功能强。C++中的type_info只能以字符串的形式显示一个类型的名字，而不能创建那个类型的对象。

5.7.2 捕获异常

我们将在第11章中全面地讲述异常处理机制，但现在时常遇到一些方法需要抛出异常。

当程序运行过程中发生错误时，就会"抛出异常"。抛出异常比终止程序要灵活得多，这是因为可以提供一个"捕获"异常的处理器（handler）对异常情况进行处理。

如果没有提供处理器，程序就会终止，并在控制台上打印出一条信息，其中给出了异常的类型。可能在前面已经看到过一些异常报告，例如，偶然使用了null引用或者数组越界等。

异常有两种类型：*未检查异常*和*已检查异常*。对于已检查异常，编译器将会检查是否提供了处理器。然而，有很多常见的异常，例如，访问null引用，都属于未检查异常。编译器不会查看是否为这些错误提供了处理器。毕竟，应该精心地编写代码来避免这些错误的发生，而不要将精力花在编写异常处理器上。

并不是所有的错误都是可以避免的。如果竭尽全力还是发生了异常，编译器就要求提供一个处理器。Class.forName方法就是一个抛出已检查异常的例子。在第11章中，将会看到几种异常处理的策略。现在，只介绍一下如何实现最简单的处理器。

将可能抛出已检查异常的一个或多个方法调用代码放在try块中，然后在catch子句中提供处理器代码。

```
try
{
   statements that might throw exceptions
}
catch(Exception e)
{
   handler action
}
```

下面是一个示例：

```
try
{
   String name = . . .; // get class name
   Class cl = Class.forName(name); // might throw exception
   . . . // do something with cl
}
catch(Exception e)
{
   e.printStackTrace();
}
```

如果类名不存在，则将跳过try块中的剩余代码，程序直接进入catch子句（这里，利用Throwable类的printStackTrace方法打印出栈的轨迹。Throwable是Exception类的超类）。如果try块中没有抛出任何异常，那么会跳过catch子句的处理器代码。

对于已检查异常，只需要提供一个异常处理器。可以很容易地发现会抛出已检查异常的方法。如果调用了一个抛出已检查异常的方法，而又没有提供处理器，编译器就会给出错误报告。

API java.lang.Class 1.0

- static Class forName(String className)

  返回描述类名为className的Class对象。
- Object newInstance()

  返回这个类的一个新实例。

API java.lang.reflect.Constructor 1.1

- `Object newInstance(Object[] args)`

  构造一个这个构造器所属类的新实例。

  参数：args　　这是提供给构造器的参数。有关如何提供参数的详细情况请参看反射的论述。

API java.lang.Throwable 1.0

- `void printStackTrace()`

  将Throwable对象和栈的轨迹输出到标准错误流。

5.7.3 利用反射分析类的能力

下面简要地介绍一下反射机制最重要的内容——检查类的结构。

在java.lang.reflect包中有三个类Field、Method和Constructor分别用于描述类的域、方法和构造器。这三个类都有一个叫做getName的方法，用来返回项目的名称。Field类有一个getType方法，用来返回描述域所属类型的Class对象。Method和Constructor类有能够报告参数类型的方法，Method类还有一个可以报告返回类型的方法。这三个类还有一个叫做getModifiers的方法，它将返回一个整型数值，用不同的位开关描述public和static这样的修饰符使用状况。另外，还可以利用java.lang.reflect包中的Modifier类的静态方法分析getModifiers返回的整型数值。例如，可以使用Modifier类中的isPublic、isPrivate或isFinal判断方法或构造器是否是public、private或final。我们需要做的全部工作就是调用Modifier类的相应方法，并对返回的整型数值进行分析，另外，还可以利用Modifier.toString方法将修饰法打印出来。

Class类中的getFields、getMethods和getConstructors方法将分别返回类提供的public域、方法和构造器数组，其中包括超类的公有成员。Class类的getDeclareFields、getDeclareMethods和getDeclaredConstructors方法将分别返回类中声明的全部域、方法和构造器，其中包括私有和受保护成员，但不包括超类的成员。

例5-6显示了如何打印一个类的全部信息的方法。这个程序将提醒用户输入类名，然后输出类中所有的方法和构造器的签名，以及全部域名。假如用户输入

```
java.lang.Double
```

程序将会输出：

```
public class java.lang.Double extends java.lang.Number
{
   public java.lang.Double(java.lang.String);
   public java.lang.Double(double);

   public int hashCode();
   public int compareTo(java.lang.Object);
   public int compareTo(java.lang.Double);
   public boolean equals(java.lang.Object);
   public java.lang.String toString();
   public static java.lang.String toString(double);
   public static java.lang.Double valueOf(java.lang.String);
   public static boolean isNaN(double);
   public boolean isNaN();
```

```
    public static boolean isInfinite(double);
    public boolean isInfinite();
    public byte byteValue();
    public short shortValue();
    public int intValue();
    public long longValue();
    public float floatValue();
    public double doubleValue();
    public static double parseDouble(java.lang.String);
    public static native long doubleToLongBits(double);
    public static native long doubleToRawLongBits(double);
    public static native double longBitsToDouble(long);

    public static final double POSITIVE_INFINITY;
    public static final double NEGATIVE_INFINITY;
    public static final double NaN;
    public static final double MAX_VALUE;
    public static final double MIN_VALUE;
    public static final java.lang.Class TYPE;
    private double value;
    private static final long serialVersionUID;
}
```

值得注意的是：这个程序可以分析Java解释器能够加载的任何类，而不仅仅是编译程序时可以使用的类。在下一章中，还将使用这个程序查看Java编译器自动生成的内部类。

例5-6 ReflectionTest.java

```
import java.util.*;
import java.lang.reflect.*;

/**
 * This program uses reflection to print all features of a class.
 * @version 1.1 2004-02-21
 * @author Cay Horstmann
 */
public class ReflectionTest
{
   public static void main(String[] args)
   {
      // read class name from command line args or user input
      String name;
      if (args.length > 0) name = args[0];
      else
      {
         Scanner in = new Scanner(System.in);
         System.out.println("Enter class name (e.g. java.util.Date): ");
         name = in.next();
      }

      try
      {
         // print class name and superclass name (if != Object)
         Class cl = Class.forName(name);
         Class supercl = cl.getSuperclass();
         String modifiers = Modifier.toString(cl.getModifiers());
         if (modifiers.length() > 0) System.out.print(modifiers + " ");
```

```
         System.out.print("class " + name);
         if (supercl != null && supercl != Object.class) System.out.print(" extends "
               + supercl.getName());

         System.out.print("\n{\n");
         printConstructors(cl);
         System.out.println();
         printMethods(cl);
         System.out.println();
         printFields(cl);
         System.out.println("}");
      }
      catch (ClassNotFoundException e)
      {
         e.printStackTrace();
      }
      System.exit(0);
   }

   /**
    * Prints all constructors of a class
    * @param cl a class
    */
   public static void printConstructors(Class cl)
   {
      Constructor[] constructors = cl.getDeclaredConstructors();

      for (Constructor c : constructors)
      {
         String name = c.getName();
         System.out.print("   ");
         String modifiers = Modifier.toString(c.getModifiers());
         if (modifiers.length() > 0) System.out.print(modifiers + " ");
         System.out.print(name + "(");

         // print parameter types
         Class[] paramTypes = c.getParameterTypes();
         for (int j = 0; j < paramTypes.length; j++)
         {
            if (j > 0) System.out.print(", ");
            System.out.print(paramTypes[j].getName());
         }
         System.out.println(");");
      }
   }

   /**
    * Prints all methods of a class
    * @param cl a class
    */
   public static void printMethods(Class cl)
   {
      Method[] methods = cl.getDeclaredMethods();

      for (Method m : methods)
      {
         Class retType = m.getReturnType();
```

```
            String name = m.getName();

            System.out.print("   ");
            // print modifiers, return type, and method name
            String modifiers = Modifier.toString(m.getModifiers());
            if (modifiers.length() > 0) System.out.print(modifiers + " ");
            System.out.print(retType.getName() + " " + name + "(");

            // print parameter types
            Class[] paramTypes = m.getParameterTypes();
            for (int j = 0; j < paramTypes.length; j++)
            {
               if (j > 0) System.out.print(", ");
               System.out.print(paramTypes[j].getName());
            }
            System.out.println(");");
         }
      }

      /**
       * Prints all fields of a class
       * @param cl a class
       */
      public static void printFields(Class cl)
      {
         Field[] fields = cl.getDeclaredFields();

         for (Field f : fields)
         {
            Class type = f.getType();
            String name = f.getName();
            System.out.print("   ");
            String modifiers = Modifier.toString(f.getModifiers());
            if (modifiers.length() > 0) System.out.print(modifiers + " ");
            System.out.println(type.getName() + " " + name + ";");
         }
      }
   }
```

API java.lang.Class 1.0

- `Field[] getFields()` 1.1
- `Filed[] getDeclaredFields()` 1.1

  getFields方法将返回一个包含Field对象的数组，这些对象记录了这个类或其超类的公有域。getDeclaredField方法也将返回包含Field对象的数组，这些对象记录了这个类的全部域。如果类中没有域，或者Class对象描述的是基本类型或数组类型，这些方法将返回一个长度为0的数组。

- `Method[] getMethods()` 1.1
- `Method[] getDeclareMethods()` 1.1

  返回包含Method对象的数组：getMethods将返回所有的公有方法，包括从超类继承来的公有方法；getDeclaredMethods返回这个类或接口的全部方法，但不包括由超类继承了的

方法。

- Constructor[] getConstructors() 1.1
- Constructor[] getDeclaredConstructors() 1.1

  返回包含Constructor对象的数组，其中包含了Class对象所描述的类的所有公有构造器（get Constructors）或所有构造器（getDeclaredConstructors）。

API java.lang.reflect.Field 1.1

API java.lang.reflect.Method 1.1

API java.lang.reflect.Constructor 1.1

- Class getDeclaringClass()

  返回一个用于描述类中定义的构造器、方法或域的Class对象。

- Class[] getExceptionTypes()（在Constructor和Method类中）

  返回一个用于描述方法抛出的异常类型的Class对象数组。

- int getModifiers()

  返回一个用于描述构造器、方法或域的修饰符的整型数值。使用Modifier类中的这个方法可以分析这个返回值。

- String getName()

  返回一个用于描述构造器、方法或域名的字符串。

- Class[] getParameterTypes()（在Constructor和Method类中）

  返回一个用于描述参数类型的Class对象数组。

- Class getReturnType()（在Method类中）

  返回一个用于描述返回类型的Class对象。

API java.lang.reflect.Modifier 1.1

- static String toString(int modifiers)

  返回对应modifiers位设置的修饰符的字符串表示。

- static boolean isAbstract(int modifiers)
- static boolean isFinal(int modifiers)
- static boolean isInterface(int modifiers)
- static boolean isNative(int modifiers)
- static boolean isPrivate(int modifiers)
- static boolean isProtected(int modifiers)
- static boolean isPublic(int modifiers)
- static boolean isStatic(int modifiers)
- static boolean isStrict(int modifiers)
- static boolean isSynchronized(int modifiers)

• static boolean isVolatile(int modifiers)

这些方法将检测方法名中对应的修饰符在modifiers值中的位。

5.7.4 在运行时使用反射分析对象

从前面一节中，已经知道如何查看任意对象的数据域名称和类型：

• 获得对应的Class对象。

• 通过Class对象调用getDeclaredFields。

本节，将进一步查看数据域的实际内容。当然，在编写程序时，如果知道想要查看的域名和类型，查看指定的域是一件很容易的事情。而利用反射机制可以查看在编译时还不清楚的对象域。

查看对象域的关键方法是Field类中的get方法。如果f是一个Field类型的对象（例如，通过getDeclaredFields得到的对象），obj是某个包含f域的类的对象，f.get(obj) 将返回一个对象，其值为obj域的当前值。这样说起来显得有点抽象，这里看一看下面这个示例的运行。

```
Employee harry = new Employee("Harry Hacker", 35000, 10, 1, 1989);
Class cl = harry.getClass();
   // the class object representing Employee
Field f = cl.getDeclaredField("name");
   // the name field of the Employee class
Object v = f.get(harry);
   // the value of the name field of the harry object
   // i.e., the String object "Harry Hacker"
```

实际上，这段代码存在一个问题。由于name是一个私有域，所以get方法将会抛出一个IllegalAccessException。只有利用get方法才能得到可访问域的值。除非拥有访问权限，否则Java安全机制只允许查看任意对象有哪些域，而不允许读取它们的值。

反射机制的默认行为受限于Java的访问控制。然而，如果一个Java程序没有受到安全管理器的控制，就可以覆盖访问控制。为了达到这个目的，需要调用Field、Method或Constructor对象的setAccessible方法。例如，

```
f.setAccessible(true); // now OK to call f.get(harry);
```

setAccessible方法是AccessibleObject类中的一个方法，它是Field、Method和Constructor类的公共超类。这个特性是为调试、持久存储和相似机制提供的。本书稍后将利用它编写一个通用的toString方法。

get方法还有一个需要解决的问题。name域是一个String，因此把它作为Object返回没有什么问题。但是，假定我们想要查看salary域。它属于double类型，而Java中数值类型不是对象。要想解决这个问题，可以使用Field类中的getDouble方法，也可以调用get方法，此时，反射机制将会自动地将这个域值打包到相应的对象包装器中，这里将打包成Double。

当然，可以获得就可以设置。调用f.set(obj, value) 可以将obj对象的f域设置成新值。

例5-7显示了如何编写一个可供任意类使用的通用toString方法。其中使用getDeclaredFileds获得所有的数据域，然后使用setAccessible将所有的域设置为可访问的。对于每个域，获得了名字和值。例5-7递归调用toString方法，将每个值转换成字符串。

```
class ObjectAnalyzer
{
  public String toString(Object obj)
  {
      Class cl = obj.getClass();
      . . .
      String r = cl.getName();
      // inspect the fields of this class and all superclasses
      do
      {
         r += "[";
         Field[] fields = cl.getDeclaredFields();
         AccessibleObject.setAccessible(fields, true);
         // get the names and values of all fields
         for (Field f : fields)
         {
            if (!Modifier.isStatic(f.getModifiers()))
            {
               if (!r.endsWith("[")) r += ","
               r += f.getName() + "=";
               try
               {
                  Object val = f.get(obj);
                  r += toString(val);
               }
               catch (Exception e) { e.printStackTrace(); }
            }
         }
         r += "]";
         cl = cl.getSuperclass();
      }
      while (cl != null);
      return r;
  }
  . . .
}
```

在例5-7的全部代码中，需要解释几个复杂的问题。循环引用将有可能导致无限递归。因此，ObjectAbalyzer将保存已经被访问过的对象。另外，为了能够查看数组内部，需要采用一种不同的方式。有关这种方式的具体内容将在下一节中详细地论述。

可以使用toString方法查看任意对象的内部信息。例如，下面这个调用

```
ArrayList<Integer> squares = new ArrayList<Integer>();
for (int i = 1; i <= 5; i++) squares.add(i * i);
System.out.println(new ObjectAnalyzer().toString(squares));
```

将会产生下面的打印结果：

```
java.util.ArrayList[elementData=class java.lang.Object[]{java.lang.Integer[value=1][][],
java.lang.Integer[value=4][][],java.lang.Integer[value=9][][],java.lang.Integer[value=16][][],
java.lang.Integer[value=25][][],null,null,null,null,null},size=5][modCount=5][][]
```

还可以使用通用的toString方法实现自己类中的toString方法，如下所示：

```
public String toString()
{
   return new ObjectAnalyzer().toString(this);
}
```

这是一种公认的提供toString方法的手段，在编写程序时会发现，它是非常有用的。

例5-7 ObjectAnalyzerTest.java

```
import java.lang.reflect.*;
import java.util.*;

/**
 * This program uses reflection to spy on objects.
 * @version 1.11 2004-02-21
 * @author Cay Horstmann
 */
public class ObjectAnalyzerTest
{
   public static void main(String[] args)
   {
      ArrayList<Integer> squares = new ArrayList<Integer>();
      for (int i = 1; i <= 5; i++)
         squares.add(i * i);
      System.out.println(new ObjectAnalyzer().toString(squares));
   }
}

class ObjectAnalyzer
{
   /**
    * Converts an object to a string representation that lists all fields.
    * @param obj an object
    * @return a string with the object's class name and all field names and
    * values
    */
   public String toString(Object obj)
   {
      if (obj == null) return "null";
      if (visited.contains(obj)) return "...";
      visited.add(obj);
      Class cl = obj.getClass();
      if (cl == String.class) return (String) obj;
      if (cl.isArray())
      {
         String r = cl.getComponentType() + "[]{";
         for (int i = 0; i < Array.getLength(obj); i++)
         {
            if (i > 0) r += ",";
            Object val = Array.get(obj, i);
            if (cl.getComponentType().isPrimitive()) r += val;
            else r += toString(val);
         }
         return r + "}";
      }

      String r = cl.getName();
      // inspect the fields of this class and all superclasses
      do
      {
         r += "[";
         Field[] fields = cl.getDeclaredFields();
```

```
            AccessibleObject.setAccessible(fields, true);
            // get the names and values of all fields
            for (Field f : fields)
            {
               if (!Modifier.isStatic(f.getModifiers()))
               {
                  if (!r.endsWith("[")) r += ",";
                  r += f.getName() + "=";
                  try
                  {
                     Class t = f.getType();
                     Object val = f.get(obj);
                     if (t.isPrimitive()) r += val;
                     else r += toString(val);
                  }
                  catch (Exception e)
                  {
                     e.printStackTrace();
                  }
               }
            }
            r += "]";
            cl = cl.getSuperclass();
         }
         while (cl != null);

         return r;
      }

   private ArrayList<Object> visited = new ArrayList<Object>();
}
```

API java.lang.reflect.AccessibleObject 1.2

- `void setAccessible(boolean flag)`
  为反射对象设置可访问标志。flag为true表明屏蔽Java语言的访问检查，使得对象的私有属性也可以被查询和设置。
- `boolean isAccessible()`
  返回反射对象的可访问标志的值。
- `static void setAccessible(AccessibleObject[] array,boolean flag)`
  是一种设置对象数组可访问标志的快捷方法。

API java.lang.Class 1.1

- `Field getField(String name)`
- `Field[] getField()`
  返回指定名称的公有域，或包含所有域的数组。
- `Field getDeclaredField(String name)`
- `Field[] getDeclaredFields()`

返回类中声明的给定名称的域，或者包含声明的全部域的数组。

API java.lang.reflect.Field 1.1

- `Object get(Object obj)`

  返回obj对象中用Field对象表示的域值。
- `void set(Object obj,Object newValue)`

  用newValue设置Obj对象中用Field对象表示的域。

5.7.5 使用反射编写泛型数组代码

java.lang.reflect包中的Array类允许动态地创建数组。例如，将这个特性应用到第3章中讲到的arrayCopy方法时，可以在保留当前数组内容的同时动态地扩展现有数组。

这里要解决一个十分具有代表性的问题。假设有一个元素为某种类型且已经被填满数组。现在希望扩展它的长度，但不想手工地编写那些扩展、拷贝元素的代码，而是想编写一个用于扩展数组的通用方法：

```
Employee[] a = new Employee[100];
. . .
// array is full
a = (Employee[]) arrayGrow(a);
```

如何编写这样一个通用的方法呢？正好能够将Employee[ ]数组转换为Object[ ]数组，这让人感觉很有希望。下面试着编写一个通用的方法，其功能是将数组扩展到10%+10个元素（这是因为对于小型数组来说，扩展10%显得有点少）。

```
static Object[] badArrayGrow(Object[] a) // not useful
{
   int newLength = a.length * 11 / 10 + 10;
   Object[] newArray = new Object[newLength];
   System.arraycopy(a, 0, newArray, 0, a.length);
   return newArray;
}
```

然而，在实际使用结果数组时会遇到一个问题。这段代码返回的数组类型是对象数组（Object[ ]）类型，这是由于使用下面这行代码创建的数组：

```
new Object[newLength]
```

一个对象数组不能转换成雇员数组（Employee[ ]）。如果这样做，则在运行时Java将会产生ClassCastException异常。前面已经看到，Java数组会记住每个元素的类型，即创建数组时new表达式中使用的元素类型。将一个Employee[ ]临时地转换成Object[ ]数组，然后再把它转换回来是可以的，但一个从开始就是Object[ ]的数组却永远不能转换成Employee[ ]数组。为了编写这类通用的数组代码，需要能够创建与原数组类型相同的新数组。为此，需要java.lang.reflect包中Array类的一些方法。其中最关键的是Array类中的静态方法newInstance，它能够构造新数组。在调用它时必须提供两个参数，一个是数组的元素类型，一个是数组的长度。

```
Object newArray = Array.newInstance(componentType, newLength);
```

为了能够实际地运行，需要获得新数组的长度和元素类型。

可以通过调用Array.getLength(a)获得数组的长度，也可以通过Array类的静态getLength方法的返回值得到任意数组的长度。而要获得新数组元素类型，就需要进行以下工作：

1）首先获得a数组的类对象。

2）确认它是一个数组。

3）使用Class类（只能定义表示数组的类对象）的getComponentType方法确定数组对应的类型。

为什么getLength是Array的方法，而getComponentType是Class的方法呢？我们也不清楚。反射方法的分类有时确实显得有点古怪。下面是这段代码：

```
static Object goodArrayGrow(Object a) // useful
{
   Class cl = a.getClass();
   if (!cl.isArray()) return null;
   Class componentType = cl.getComponentType();
   int length = Array.getLength(a);
   int newLength = length * 11 / 10 + 10;
   Object newArray = Array.newInstance(componentType, newLength);
   System.arraycopy(a, 0, newArray, 0, length);
   return newArray;
}
```

请注意，arrayGrow方法可以用来扩展任意类型的数组，而不仅是对象数组。

```
int[] a = { 1, 2, 3, 4 };
a = (int[]) goodArrayGrow(a);
```

为了能够实现上述操作，应该将goodArrayGrow的参数声明为Object类型，而不要声明为对象型数组（Object[]）。整型数组类型int[]可以被转换成Object，但不能转换成对象数组。

例5-8显示了两个扩展数组的方法。请注意，将badArrayGrow的返回值进行类型转换将会抛出一个异常。

✔ **注释**：这段程序显示了通过反射，数组的工作过程。如果只希望扩大数组，利用Arrays类的copyOf方法就可以。

```
Employee[] a = new Employee[100];
. . .
// array is full
a = Arrays.copyOf(a, a.length * 11 / 10 + 10);
```

例5-8 ArrayGrowTest.java

```
import java.lang.reflect.*;

/**
 * This program demonstrates the use of reflection for manipulating arrays.
 * @version 1.01 2004-02-21
 * @author Cay Horstmann
 */
public class ArrayGrowTest
{
   public static void main(String[] args)
   {
```

```
      int[] a = { 1, 2, 3 };
      a = (int[]) goodArrayGrow(a);
      arrayPrint(a);

      String[] b = { "Tom", "Dick", "Harry" };
      b = (String[]) goodArrayGrow(b);
      arrayPrint(b);

      System.out.println("The following call will generate an exception.");
      b = (String[]) badArrayGrow(b);
   }

   /**
    * This method attempts to grow an array by allocating a new array and copying all elements.
    * @param a the array to grow
    * @return a larger array that contains all elements of a. However, the returned array has
    * type Object[], not the same type as a
    */
   static Object[] badArrayGrow(Object[] a)
   {
      int newLength = a.length * 11 / 10 + 10;
      Object[] newArray = new Object[newLength];
      System.arraycopy(a, 0, newArray, 0, a.length);
      return newArray;
   }

   /**
    * This method grows an array by allocating a new array of the same type and
    * copying all elements.
    * @param a the array to grow. This can be an object array or a primitive
    * type array
    * @return a larger array that contains all elements of a.
    */
   static Object goodArrayGrow(Object a)
   {
      Class cl = a.getClass();
      if (!cl.isArray()) return null;
      Class componentType = cl.getComponentType();
      int length = Array.getLength(a);
      int newLength = length * 11 / 10 + 10;

      Object newArray = Array.newInstance(componentType, newLength);
      System.arraycopy(a, 0, newArray, 0, length);
      return newArray;
   }

   /**
    * A convenience method to print all elements in an array
    * @param a the array to print. It can be an object array or a primitive type array
    */
   static void arrayPrint(Object a)
   {
      Class cl = a.getClass();
      if (!cl.isArray()) return;
      Class componentType = cl.getComponentType();
      int length = Array.getLength(a);
```

```
        System.out.print(componentType.getName() + "[" + length + "] = { ");
        for (int i = 0; i < Array.getLength(a); i++)
           System.out.print(Array.get(a, i) + " ");
        System.out.println("}");
     }
  }
```

API java.lang.reflect.Array 1.1

- `static Object get(Objkect array,int index)`
- `static xxx getXxx(Object array,int index)`

  (xxx是boolean、byte、char、double、float、int、long、short之中的一种基本类型。）这些方法将返回存储在给定位置上的给定数组的内容。
- `static void set(Object array,int index,Object newValue)`
- `static setXxx(Object array,int index,xxx newValue)`

  (xxx是boolean、byte、char、double、float、int、long、short之中的一种基本类型。）这些方法将一个新值存储到给定位置上的给定数组中。
- `static int getLength(Object array)`

  返回数组的长度。
- `static Object newInstance(Class componentType,int length)`
- `static Object newInstance(Class componentType,int[] lengths)`

  返回一个具有给定类型、给定维数的新数组。

5.7.6 方法指针

从表面上看，Java没有提供方法指针，即将一个方法的存储地址传给另外一个方法，以便第二个方法能够随后调用它。事实上，Java的设计者曾说过：方法指针是很危险的，并且常常会带来隐患。他们认为Java提供的*接口*（interface）（将在下一章讨论）是一种更好的解决方案。然而，在Java 1.1中方法指针已经作为反射包的（也许是）副产品出现了。

注释：*微软公司为自己的非标准Java语言J++（以及后来的C#）增加了另一种被称为委托（delegate）的方法指针类型，它与本节讨论的Method类不同。然而，在下一章中讨论的内部类比委托更加有用。*

为了能够看到方法指针的工作过程，先回忆一下利用Field类的get方法查看对象域的过程。与之类似，在Method类中有一个invoke方法，它允许调用包装在当前Method对象中的方法。invoke方法的签名是：

```
Object invoke(Object obj, Object... args)
```

第一个参数是隐式参数，其余的对象提供了显式参数（在Java SE 5.0以前的版本中，必须传递一个对象数组，如果没有显式参数就传递一个null)。

对于静态方法，第一个参数可以被忽略，即可以将它设置为null。

例如，假设用ml代表Employee类的getName方法，下面这条语句显示了如何调用这个方法：

```
String n = (String) m1.invoke(harry);
```

如果参数或返回类型不是类而是基本类型，那么在调用Field类的get和set方法时会存在一些问题。需要依靠自动打包功能将其打包，或者在Java SE 5.0之前将基本类型打包成对应的包装器。

相反地，如果返回类型是一种基本类型，则invoke方法将返回包装器类型。例如，假设m2代表Employee类的getSalary方法，返回的实际类型是Double，因此必须相应地进行类型转换。对于Java SE 5.0来说，这项操作可以由自动拆包来完成。

```
double s = (Double) m2.invoke(harry);
```

如何得到Method对象呢？当然，可以通过调用getDeclareMethods方法，然后对返回的Method对象数组进行查找，直到发现想要的方法为止。也可以通过调用Class类中的getMethod方法得到想要的方法。它与getField方法类似。getField方法根据表示域名的字符串，返回一个Field对象。然而，有可能存在若干个相同名字的方法，因此要格外地小心，以确保能够准确地得到想要的那个方法。有鉴于此，还必须提供想要的方法的参数类型。getMethod的签名是：

```
Method getMethod(String name, Class... parameterTypes)
```

例如，下面说明了如何获得Employee类的getName方法和raiseSalary方法的方法指针。

```
Method m1 = Employee.class.getMethod("getName");
Method m2 = Employee.class.getMethod("raiseSalary", double.class);
```

（对于Java SE 5.0以前的版本，必须将Class对象包装到一个数组中，如果没有参数，需要给出一个null）。

到此为止，读者已经学习了使用Method对象的规则。下面看一下如何将它们组织在一起。例5-9是一个打印诸如Math.sqrt、Math.sin这样的数学函数值表的程序。打印的结果如下所示：

```
public static native double java.lang.Math.sqrt(double)
      1.0000 |      1.0000
      2.0000 |      1.4142
      3.0000 |      1.7321
      4.0000 |      2.0000
      5.0000 |      2.2361
      6.0000 |      2.4495
      7.0000 |      2.6458
      8.0000 |      2.8284
      9.0000 |      3.0000
     10.0000 |      3.1623
```

当然，这段打印数学函数表格的代码与具体打印的数学函数无关。

```
double dx = (to - from) / (n - 1);
for (double x = from; x <= to; x += dx)
{
   double y = (Double) f.invoke(null, x);
   System.out.printf("%10.4f | %10.4f%n", x, y);
}
```

在这里，f是一个Method类型的对象。由于正在调用的方法是一个静态方法，所以invoke的第一个参数是null。

为了将Math.sqrt函数表格化，需要将f设置为：

```
Math.class.getMethod("sqrt", double.class)
```

这是Math类中的一个方法，通过参数向它提供了一个函数名sqrt和一个double类型的参数。

例5-9给出了通用制表和两个测试程序的全部代码。

例5-9 MethodPointerTest.java

```
import java.lang.reflect.*;

/**
 * This program shows how to invoke methods through reflection.
 * @version 1.1 2004-02-21
 * @author Cay Horstmann
 */
public class MethodPointerTest
{
   public static void main(String[] args) throws Exception
   {
      // get method pointers to the square and sqrt methods
      Method square = MethodPointerTest.class.getMethod("square", double.class);
      Method sqrt = Math.class.getMethod("sqrt", double.class);

      // print tables of x- and y-values

      printTable(1, 10, 10, square);
      printTable(1, 10, 10, sqrt);
   }

   /**
    * Returns the square of a number
    * @param x a number
    * @return x squared
    */
   public static double square(double x)
   {
      return x * x;
   }

   /**
    * Prints a table with x- and y-values for a method
    * @param from the lower bound for the x-values
    * @param to the upper bound for the x-values
    * @param n the number of rows in the table
    * @param f a method with a double parameter and double return value
    */
   public static void printTable(double from, double to, int n, Method f)
   {
      // print out the method as table header
      System.out.println(f);

      double dx = (to - from) / (n - 1);

      for (double x = from; x <= to; x += dx)
      {
         try
         {
            double y = (Double) f.invoke(null, x);
            System.out.printf("%10.4f | %10.4f%n", x, y);
```

```
        }
        catch (Exception e)
        {
           e.printStackTrace();
        }
     }
  }
}
```

上述程序清楚地表明，可以使用method对象实现C（或C#中的委派）语言中函数指针的所有操作。同C一样，这种程序设计风格并不太简便，出错的可能性也比较大。如果在调用方法的时候提供了一个错误的参数，那么invoke方法将会抛出一个异常。

另外，invoke的参数和返回值必须是Object类型的。这就意味着必须进行多次的类型转换。这样做将会使编译器错过检查代码的机会。因此，等到测试阶段才会发现这些错误，找到并改正它们将会更加困难。不仅如此，使用反射获得方法指针的代码要比仅仅直接调用方法明显慢一些。

有鉴于此，建议仅在必要的时候才使用Method对象，而最好使用接口和内部类（第6章中介绍）。特别要重申：建议Java开发者不要使用Method对象的回调功能。使用接口进行回调（第6章介绍）会使得代码的执行速度更快，更易于维护。

API `java.lang.reflect.Method` 1.1

- `public Object invoke(Object implicitParameter,Object[] explicitParamenters)`
  调用这个对象所描述的方法，传递给定参数，并返回方法的返回值。对于静态方法，把null作为隐式参数传递。在使用包装器传递基本类型的值时，基本类型的返回值必须是未包装的。

5.8 继承设计的技巧

下面给出一些对设计继承关系很有帮助的建议，以此结束本章的内容。

1）将公共操作和域放在超类。

这就是为什么将姓名域放在person类中，而没有将它放在Employee和Student类中的原因。

2）不要使用受保护的域。

有些程序员认为，将大多数的实例域定义为protected是一个不错的主意，只有这样，子类才能够在需要的时候直接访问它们。然而，protected机制并不能够带来更好的保护，其原因主要有两点。第一，子类集合是无限制的，任何一个人都能够由某个类派生一个子类，并编写代码以直接访问protected的实例域，从而破坏了封装性。第二，在Java程序设计语言中，在同一个包中的所有类都可以访问proteced域，而不管它是否为这个类的子类。

3）使用继承实现“is-a”关系。

使用继承很容易达到节省代码的目的，但有时候也被人们滥用了。例如，假设需要定义一个钟点工类。钟点工的信息包含姓名和雇佣日期，但是没有薪水。他们按小时计薪，并且不会因为拖延时间而获得加薪。这似乎在诱导人们由Employee派生出子类Contractor，然后再增加

一个hourlyWage域。

```
class Contractor extends Employee
{ . . .
   private double hourlyWage;
}
```

这并不是一个好主意。因为这样一来，每个钟点工对象中都包含了薪水和计时工资这两个域。在实现打印支票或税单方法的时候，会带来无尽的麻烦，并且与不采用继承，会多写很多代码。

钟点工与雇员之间不属于“is-a”关系。钟点工不是特殊的雇员。

4）*除非所有继承的方法都有意义，否则不要使用继承。*

假设想编写一个Holiday类。毫无疑问，每个假日也是一日，并且一日可以用GregorianCalendar类的实例表示，因此可以使用继承。

```
class Holiday extends GregorianCalendar { . . . }
```

很遗憾，在继承的操作中，假日集不是封闭的。在GregorianCalendar中有一个公有方法add，可以将假日转换成非假日：

```
Holiday christmas;
christmas.add(Calendar.DAY_OF_MONTH, 12);
```

因此，继承对于这个例子来说并不太适宜。

5）*在覆盖方法时，不要改变预期的行为。*

置换原则不仅应用于语法，而且也可以应用于行为，这似乎更加重要。在覆盖一个方法的时候，不应该毫无原由地改变行为的内涵。就这一点而言，编译器不会提供任何帮助，即编译器不会检查重新定义的方法是否有意义。例如，可以重定义Holiday类中add方法“修正”原方法的问题，或什么也不做，或抛出一个异常，或继续到下一个假日。然而这些都违反了置换原则。语句序列

```
int d1 = x.get(Calendar.DAY_OF_MONTH);
x.add(Calendar.DAY_OF_MONTH, 1);
int d2 = x.get(Calendar.DAY_OF_MONTH);
System.out.println(d2 - d1);
```

不管x属于GregorianCalendar类，还是属于Holiday类，执行上述语句后都应该得到预期的行为。

当然，这样可能会引起某些争议。人们可能就预期行为的含义争论不休。例如，有些人争论说，置换原则要求Manager.equals不处理bonus域，因为Employee.equals没有它。实际上，凭空讨论这些问题毫无意义。关键在于，在覆盖子类中的方法时，不要偏离最初的设计想法。

6）*使用多态，而非类型信息。*

无论什么时候，对于下面这种形式的代码

```
if (x is of type 1)
   action1(x);
else if (x is of type 2)
   action2(x);
```

都应该考虑使用多态性。

action1与action2表示的是相同的概念吗？如果是相同的概念，就应该为这个概念定义一个方法，并将其放置在两个类的超类或接口中，然后，就可以调用

```
x.action();
```

以便使用多态性提供的动态分派机制执行相应的动作。

使用多态方法或接口编写的代码比使用对多种类型进行检测的代码更加易于维护和扩展。

7）不要过多地使用反射。

反射机制使得人们可以通过在运行时查看域和方法，让人们编写出更具有通用性的程序。这种功能对于编写系统程序来说极其实用，但是通常不适于编写应用程序。反射是很脆弱的，即编译器很难帮助人们发现程序中的错误。任何错误只能在运行时才被发现，并导致异常。

第6章　接口与内部类

- ▲ 接口
- ▲ 对象克隆
- ▲ 接口与回调
- ▲ 内部类
- ▲ 代理

到目前为止，读者已经学习了Java面向对象程序设计的全部基本知识。本章将开始介绍几种常用的高级技术。这些内容可能不太容易理解，但一定要掌握它们，以便完善自己的Java工具箱。

首先，介绍一下*接口*（interface）技术，这种技术主要用来描述类具有什么功能，而并不给出每个功能的具体实现。一个类可以*实现*（implement）一个或多个接口，并在需要接口的地方，随时使用实现了相应接口的对象。了解接口以后，再继续看一下克隆对象（有时又称为深拷贝）。对象的克隆是指创建一个新对象，且新对象的状态与原始对象的状态相同。当对克隆的新对象进行修改时，不会影响原始对象的状态。

接下来，看一下*内部类*（inner class）机制。内部类定义在另外一个类的内部，其中的方法可以访问包含它们的外部类的域，这是一项比较复杂的技术。内部类技术主要用于设计具有相互协作关系的类集合。特别是在编写处理GUI事件的代码时，使用它将可以让代码看起来更加简练专业。

在本章的最后还将介绍*代理*（proxy），这是一种实现任意接口的对象。代理是一种非常专业的构造工具，它可以用来构建系统级的工具。如果是第一次学习这本书，可以先跳过这个部分。

6.1　接口

在Java程序设计语言中，接口不是类，而是对类的一组*需求描述*，这些类要遵从接口描述的统一格式进行定义。

我们经常听到服务提供商这样说：“如果类遵从某个特定接口，那么就履行这项服务”。下面给出一个具体的示例。Arrays类中的sort方法承诺可以对对象数组进行排序，但要求满足下列前提：对象所属的类必须实现了Comparable接口。

下面是Comparable接口的代码：

```
public interface Comparable
{
   int compareTo(Object other);
}
```

这就是说，任何实现Comparable接口的类都需要包含compareTo方法，并且这个方法的参数必须是一个Object对象，返回一个整型数值。

注释：在Java SE 5.0中，Comparable接口已经改进为泛型类型。

```
public interface Comparable<T>
{
   int compareTo(T other); // parameter has type T
}
```

例如，在实现Comparable<Employee>接口的类中，必须提供下列方法

```
int compareTo(Employee other)
```

也可以使用没有类型参数的“原始”Comparable类型，但必须手工地将compareTo方法的参数转换成所希望的类型。

接口中的所有方法自动地属于public。因此，在接口中声明方法时，不必提供关键字public。

当然，接口中还有一个没有明确说明的附加要求：在调用x.compareTo(y)的时候，这个compareTo方法必须确实比较两个对象的内容，并返回比较的结果。当x小于y时，返回一个负数；当x等于y时，返回0；否则返回一个正数。

上面这个接口只有一个方法，而有些接口可能包含多个方法。稍后可以看到，在接口中还可以定义常量。然而，更为重要的是要知道接口不能提供哪些功能。接口绝不能含有实例域，也不能在接口中实现方法。提供实例域和方法实现的任务应该由实现接口的那个类来完成。因此，可以将接口看成是没有实例域的抽象类。但是这两个概念还是有一定区别的，稍后将给出详细的解释。

现在，假设希望使用Arrays类的sort方法对Employee对象数组进行排序，Employee类就必须实现Comparable接口。

为了让类实现一个接口，通常需要下面两个步骤：

1）将类声明为实现给定的接口。

2）对接口中的所有方法进行定义。

要将类声明为实现某个接口，需要使用关键字implements：

```
class Employee implements Comparable
```

当然，这里的Employee类需要提供compareTo方法。假设希望根据雇员的薪水进行比较。如果第一个雇员的薪水低于第二个雇员的薪水就返回－1；如果相等就返回0；否则返回1。

```
public int compareTo(Object otherObject)
{
   Employee other = (Employee) otherObject;
   if (salary < other.salary) return -1;
   if (salary > other.salary) return 1;
   return 0;
}
```

警告：在接口声明中，没有将compareTo方法声明为public，这是因为在接口中的所有方法都自动地是public。不过，在实现接口时，必须把方法声明为public；否则，编译器将认为这个方法的访问属性是包可见性，即类的默认访问属性，之后编译器就会给出试图提供更弱的访问权限的警告信息。

在Java SE 5.0中，可以做得更好一些。可以将上面的实现替换为对Comparable<Employee>接口的实现。

```
class Employee implements Comparable<Employee>
{
   public int compareTo(Employee other)
   {
      if (salary < other.salary) return -1;
      if (salary > other.salary) return 1;
      return 0;
   }
   . . .
}
```

请注意，将参数Object进行类型转换总是让人感觉不太顺眼，但现在已经不见了。

提示：Comparable接口中的compareTo方法将返回一个整型数值。如果两个对象不相等，则返回一个正值或者一个负值。在对两个整数域进行比较时，这点非常有用。例如，假设每个雇员都有一个惟一整数id，并希望根据ID对雇员进行重新排序，那么就可以返回id-other.id。如果第一个ID小于另一个ID，则返回一个负值；如果两个ID相等，则返回0；否则，返回一个正值。但有一点需要注意：整数的范围不能过大，以避免造成减法运算的溢出。如果能够确信ID为非负整数，或者它们的绝对值不会超过(Integer.MAX_VALUE-1)/2，就不会出现问题。

当然，这里的相减技巧不适用于浮点值。因为在salary和other.salary很接近但又不相等的时候，它们的差经过四舍五入后有可能变成0。

现在，我们已经看到，要让一个类使用排序服务必须让它实现compareTo方法。这是理所当然的，因为要向sort方法提供对象的比较方式。但是为什么不能在Employee类直接提供一个compareTo方法，而必须实现Comparable接口呢？

主要原因在于Java程序设计语言是一种强类型（strongly typed）语言。在调用方法的时候，编译器将会检查这个方法是否存在。在sort方法中可能存在下面这样的语句：

```
if (a[i].compareTo(a[j]) > 0)
{
   // rearrange a[i] and a[j]
   . . .
}
```

为此，编译器必须确认a[i]一定有compareTo方法。如果a是一个Comparable对象的数组，就可以确保拥有compareTo方法，因为每个实现Comparable接口的类都必须提供这个方法的定义。

注释：有人认为，将Arrays类中的sort方法定义为接收一个Comparable[ ]数组就可以在使用元素类型没有实现Comparable接口的数组作为参数调用sort方法时，由编译器给出错误报告。但事实并非如此。在这种情况下，sort方法可以接收一个Object[ ]数组，并对其进行笨拙的类型转换：

```
// from the standard library--not recommended
if (((Comparable) a[i]).compareTo(a[j]) > 0)
```

```
{
   // rearrange a[i] and a[j]
   . . .
}
```

如果a[i]不属于实现了Comparable接口的类，那么虚拟机就会抛出一个异常。

例6-1给出了实现雇员数组排序的全部代码。

例6-1 EmployeeSortTest.java

```
import java.util.*;

/**
 * This program demonstrates the use of the Comparable interface.
 * @version 1.30 2004-02-27
 * @author Cay Horstmann
 */
public class EmployeeSortTest
{
   public static void main(String[] args)
   {
      Employee[] staff = new Employee[3];

      staff[0] = new Employee("Harry Hacker", 35000);
      staff[1] = new Employee("Carl Cracker", 75000);
      staff[2] = new Employee("Tony Tester", 38000);

      Arrays.sort(staff);

      // print out information about all Employee objects
      for (Employee e : staff)
         System.out.println("name=" + e.getName() + ",salary=" + e.getSalary());
   }
}

class Employee implements Comparable<Employee>
{
   public Employee(String n, double s)
   {
      name = n;
      salary = s;
   }

   public String getName()
   {
      return name;
   }

   public double getSalary()
   {
      return salary;
   }

   public void raiseSalary(double byPercent)
   {
      double raise = salary * byPercent / 100;
```

```
      salary += raise;
   }

   /**
    * Compares employees by salary
    * @param other another Employee object
    * @return a negative value if this employee has a lower salary than
    * otherObject, 0 if the salaries are the same, a positive value otherwise
    */
   public int compareTo(Employee other)
   {
      if (salary < other.salary) return -1;
      if (salary > other.salary) return 1;
      return 0;
   }

   private String name;
   private double salary;
}
```

API java.lang.Comparable<T> 1.0

- `int compareTo(T other)`

  用这个对象与other进行比较。如果这个对象小于other则返回负值；如果相等则返回0；否则返回正值。

API java.util.Arrays 1.2

- `static void sort(Object[] a)`

  使用mergesort算法对数组a中的元素进行排序。要求数组中的元素必须属于实现了Comparable接口的类，并且元素之间必须是可比较的。



注释：语言标准规定：对于任意的x和y，实现必须能够保证sgn(x.compareTo(y)) = -sgn (y.compareTo(x))。（也就是说，如果y.compareTo(x) 抛出一个异常，那么x.compareTo (y) 也应该抛出一个异常。）这里的"sgn"是一个数值的符号：如果n是负值，sgn(n)等于－1；如果n是0，sgn(n)等于0；如果n是正值，sgn(n)等于1。简单地讲，如果调换compareTo的参数，结果的符号也应该调换（而不是实际值）。

与equals方法一样，在继承过程中有可能会出现问题。

这是因为Manager扩展了Employee，而Employee实现的是Comparable<Employee>，而不是Comparable<Manager>。如果Manager覆盖了compareTo，就必须要有经理与雇员进行比较的思想准备，绝不能仅仅将雇员转换成经理。

```
class Manager extends Employee
{
   public int compareTo(Employee other)
   {
      Manager otherManager = (Manager) other; // NO
      ...
   }
```

```
    ...
}
```

这不符合“反对称”的规则。如果x是一个Employee对象，y是一个Manager对象，调用x.compareTo(y)不会抛出异常，它只是将x和y都作为雇员进行比较。但是反过来，y.compareTo(x)将会抛出一个ClassCastException。

这种情况与第5章中讨论的equals方法一样，修改的方式也一样。有两种不同的情况。

如果子类之间的比较含义不一样，那就属于不同类对象的非法比较。每个compareTo方法都应该在开始时进行下列检测：

```
if (getClass() != other.getClass()) throw new ClassCastException();
```

如果存在这样一种通用算法，它能够对两个不同的子类对象进行比较，则应该在超类中提供一个compareTo方法，并将这个方法声明为final。

例如，假设不管薪水的多少都想让经理大于雇员，像Executive和Secretary这样的子类又该怎么办呢？如果一定要按照职务排列的话，那就应该在Employee类中提供一个rank方法。每个子类覆盖rank，并实现一个比较rank值的compareTo方法。

6.1.1 接口的特性

接口不是类，尤其不能使用new运算符实例化一个接口：

```
x = new Comparable(. . .); // ERROR
```

然而，尽管不能构造接口的对象，却能声明接口的变量：

```
Comparable x; // OK
```

接口变量必须引用实现了接口的类对象：

```
x = new Employee(. . .); // OK provided Employee implements Comparable
```

接下来，如同使用instanceof检查一个对象是否属于某个特定类一样，也可以使用instance检查一个对象是否实现了某个特定的接口：

```
if (anObject instanceof Comparable) { . . . }
```

与可以建立类的继承关系一样，接口也可以被扩展。这里允许存在多条从具有较高通用性的接口到较高专用性的接口的链。例如，假设有一个称为Moveable的接口：

```
public interface Moveable
{
   void move(double x, double y);
}
```

然后，可以以它为基础扩展一个叫做Powered的接口：

```
public interface Powered extends Moveable
{
   double milesPerGallon();
}
```

虽然在接口中不能包含实例域或静态方法，但却可以包含常量。例如：

```
public interface Powered extends Moveable
{
   double milesPerGallon();
```

```
    double SPEED_LIMIT = 95; // a public static final constant
}
```

与接口中的方法都自动地被设置为public一样，接口中的域将被自动设为public static final。

✔ **注释**：可以将接口方法标记为public，将域标记为public static final。有些程序员出于习惯或提高清晰度的考虑，愿意这样做。但Java语言规范却建议不要书写这些多余的关键字，本书也采纳了这个建议。

有些接口只定义了常量，而没有定义方法。例如，在标准库中有一个SwingConstants就是这样一个接口，其中只包含NORTH、SOUTH和HORIZONTAL等常量。任何实现SwingConstants接口的类都自动地继承了这些常量，并可以在方法中直接地引用NORTH，而不必采用SwingConstants.NORTH这样的繁琐书写形式。然而，这样应用接口似乎有点偏离了接口概念的初衷，最好不要这样使用它。

尽管每个类只能够拥有一个超类，但却可以实现多个接口。这就为定义类的行为提供了极大的灵活性。例如，Java程序设计语言有一个非常重要的内置接口，称为Cloneable（将在下一节中给予详细的讨论）。如果某个类实现了这个Cloneable接口，Object类中的clone方法就可以创建类对象的一个拷贝。如果希望自己设计的类拥有克隆和比较的能力，只要实现这两个接口就可以了。

```
class Employee implements Cloneable, Comparable
```

使用逗号将实现的各个接口分隔开。

6.1.2 接口与抽象类

如果阅读了第5章中有关抽象类的内容，那就可能会产生这样一个疑问：为什么Java程序设计语言还要不辞辛苦地引入接口概念？为什么不将Comparable直接设计成如下所示的抽象类。

```
abstract class Comparable // why not?
{
   public abstract int compareTo(Object other);
}
```

然后，Employee类再直接扩展这个抽象类，并提供compareTo方法的实现：

```
class Employee extends Comparable // why not?
{
   public int compareTo(Object other) { . . . }
}
```

非常遗憾，使用抽象类表示通用属性存在这样一个问题：每个类只能扩展于一个类。假设Employee类已经扩展于一个类，例如Person，它就不能再像下面这样扩展第二个类了：

```
class Employee extends Person, Comparable // ERROR
```

但每个类可以像下面这样实现多个接口：

```
class Employee extends Person implements Comparable // OK
```

有些程序设计语言允许一个类有多个超类，例如C++。我们将此特性称为*多继承*（multiple inheritance）。而Java的设计者选择了不支持多继承，其主要原因是多继承会让语言本身变得非

常复杂（如同C++），效率也会降低（如同Eiffel）。

为了避免这类问题的出现，Java语言利用接口机制来实现多继承的大部分功能。

C++注释：C++具有多继承，随之带来了一些诸如虚基类、控制规则和横向指针类型转换等复杂特性。很少有C++程序员使用多继承，甚至有些人说：就不应该使用多继承。也有些程序员建议只对"混合"风格的继承使用多继承。在"混合"风格中，一个主要的基类描述父对象，其他的基类（因此称为混合）扮演辅助的角色。这种风格类似于Java类中从一个基类派生，然后实现若干个辅助接口。然而，在C++中，"混合"类可以添加默认的行为，而Java的接口则不行。

6.2 对象克隆

当拷贝一个变量时，原始变量与拷贝变量引用同一个对象，如图6-1所示。这就是说，改变一个变量所引用的对象将会对另一个变量产生影响。

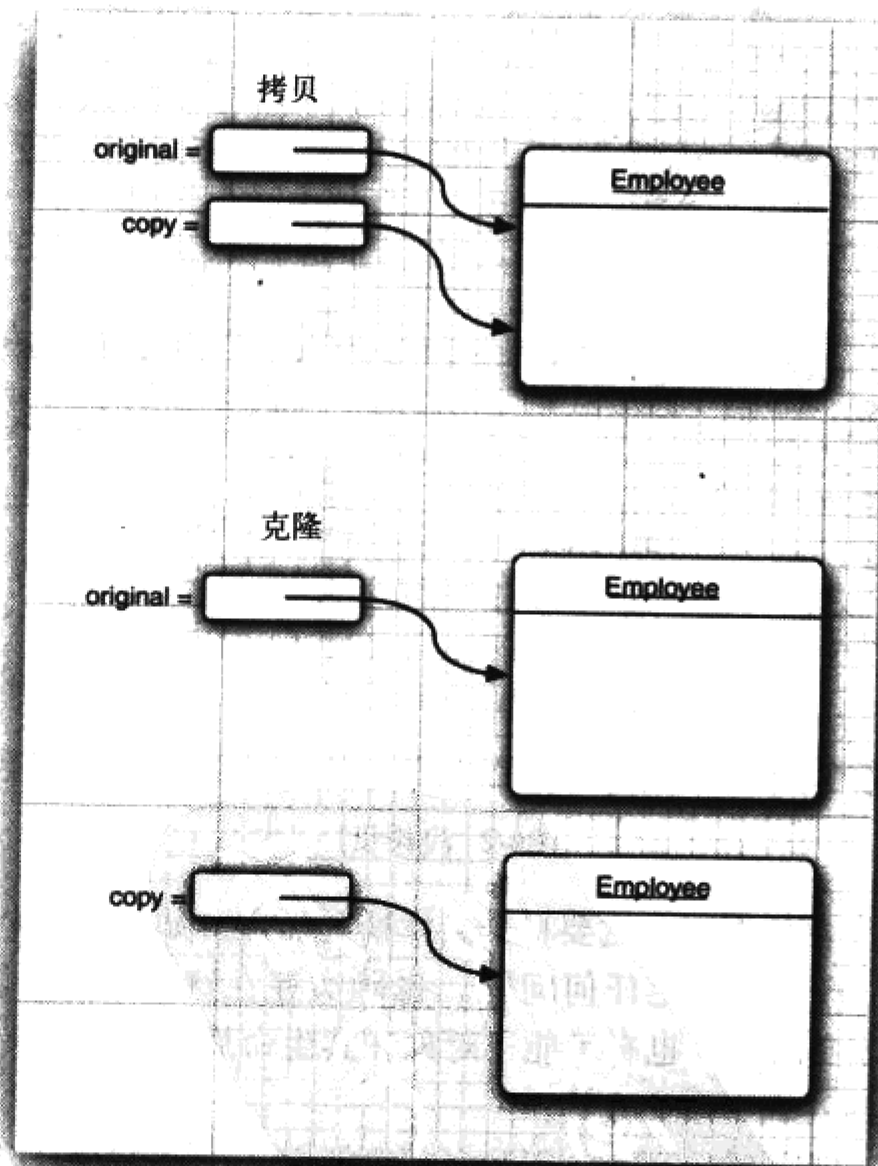


图6-1 拷贝与克隆

```
Employee original = new Employee("John Public", 50000);
Employee copy = original;
copy.raiseSalary(10); // oops--also changed original
```

如果创建一个对象的新的copy，它的最初状态与original一样，但以后将可以各自改变各自的状态，那就需要使用clone方法。

```
Employee copy = original.clone();
copy.raiseSalary(10); // OK--original unchanged
```

不过，事情并没有这么简单。clone方法是Object类的一个proteced方法，也就是说，在用户编写的代码中不能直接调用它。只有Employee类才能够克隆Employee对象。这种限制有一定的道理。这里查看一下Object类实现的clone方法。由于这个类对具体的类对象一无所知，所以只能将各个域进行对应的拷贝。如果对象中的所有数据域都属于数值或基本类型，这样拷贝域没有任何问题。但是，如果在对象中包含了子对象的引用，拷贝的结果会使得两个域引用同一个子对象，因此原始对象与克隆对象共享这部分信息。

为了能够说明这种现象，请再看一下第4章中介绍的Employee类。图6-2显示了使用Object类的clone方法克隆Employee对象的结果。可以看到，默认的克隆操作是*浅拷贝*，它并没有克隆包含在对象中的内部对象。

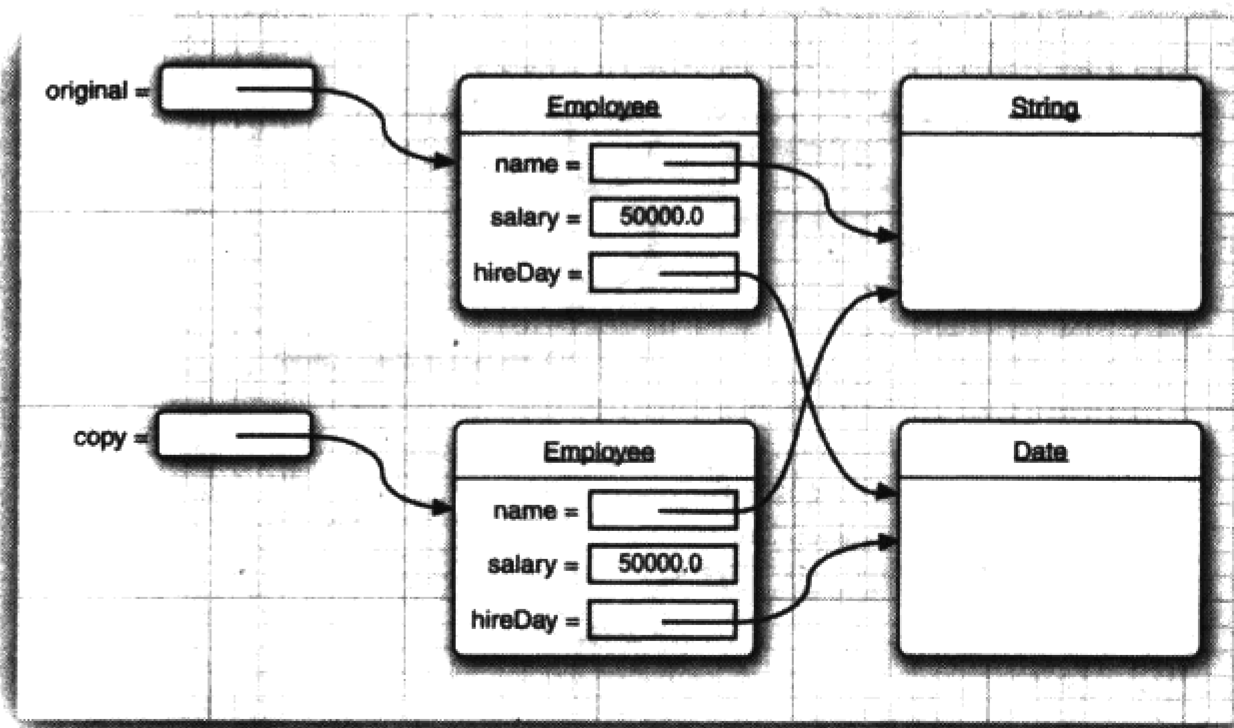


图6-2 浅拷贝

如果进行浅拷贝会发生什么呢？这要根据具体情况而定。如果原始对象与浅克隆对象共享的子对象是*不可变的*，将不会产生任何问题。也确实存在这种情形。例如，子对象属于像String类这样的不允许改变的类；也有可能子对象在其生命周期内不会发生变化，既没有更改它们的方法，也没有创建对它引用的方法。

然而，更常见的情况是子对象可变，因此必须重新定义clone方法，以便实现克隆子对象的*深拷贝*。在列举的示例中，hireDay域属于Date类，这就是一个可变的子对象。

对于每一个类，都需要做出下列判断：

1）默认的clone方法是否满足要求。

2）默认的clone方法是否能够通过调用可变子对象的clone得到修补。

3）是否不应该使用clone。

实际上，选项3是默认的。如果要选择1或2，类必须：

1）实现Cloneable接口。

2）使用public访问修饰符重新定义clone方法。

注释：在Object类中，clone方法被声明为protected，因此无法直接调用anObject.clone( )。但是，不是所有子类都可以访问受保护的方法吗？不是每个类都是Object的子类吗？值得庆幸的是，受保护访问的规则极为微妙（参阅第5章）。子类只能调用受保护的clone方法克隆它自己。为此，必须重新定义clone方法，并将它声明为public，这样才能够让所有的方法克隆对象。

在这里，Cloneable接口的出现与接口的正常使用没有任何关系。尤其是，它并没有指定clone方法，这个方法是从Object类继承而来的。接口在这里只是作为一个标记，表明类设计者知道要进行克隆处理。如果一个对象需要克隆，而没有实现Cloneable接口，就会产生一个已检验异常（checked exception）。

注释：Cloneable接口是Java提供的几个标记接口（tagging interface）之一（有些程序员将它们称为标记接口（marker interface））。我们知道，通常使用接口的目的是为了确保类实现某个特定的方法或一组特定的方法，Comparable接口就是这样一个示例。而标记接口没有方法，使用它的惟一目的是可以用instanceof进行类型检查：

```
if (obj instanceof Cloneable) . . .
```

建议在自己编写程序时，不要使用这种技术。

即使clone的默认实现（浅拷贝）能够满足需求，也应该实现Cloneable接口，将clone重定义为public，并调用super.clone( )。下面是一个示例：

```
class Employee implements Cloneable
{
   // raise visibility level to public, change return type
   public Employee clone() throws CloneNotSupportedException
   {
      return (Employee) super.clone();
   }
   . . .
}
```

注释：在Java SE 5.0以前的版本中，clone方法总是返回Object类型，而在Java SE 5.0中，允许克隆方法指定返回类型。

刚才看到的clone方法并没有在Object.clone提供的浅拷贝基础上增加任何新功能，而只是将这个方法声明为public。为了实现深拷贝，必须克隆所有可变的实例域。

下面是一个建立深拷贝clone方法的一个示例：

```
class Employee implements Cloneable
{
```

```
   . . .
   public Employee clone() throws CloneNotSupportedException
   {
      // call Object.clone()
      Employee cloned = (Employee) super.clone();

      // clone mutable fields
      cloned.hireDay = (Date) hireDay.clone()

      return cloned;
   }
}
```

只要在clone中含有没有实现Cloneable接口的对象，Object类的clone方法就会抛出一个CloneNot-SupportException异常。当然，Employee和Date类都实现了Cloneable接口，因此不会抛出异常。但是编译器并不知道这些情况，因此需要声明异常：

```
public Employee clone() throws CloneNotSupportedException
```

如果将上面这种形式替换成捕获异常呢？

```
public Employee clone()
{
   try
   {
      return super.clone();
   }
   catch (CloneNotSupportedException e) { return null; }
   // this won't happen, since we are Cloneable
}
```

这种写法比较适用于final类，否则最好还是在这个地方保留throws说明符。如果不支持克隆，子类具有抛出CloneNotSupportException异常的选择权。

必须谨慎地实现子类的克隆。例如，一旦为Employee类定义了clone方法，任何人都可以利用它克隆Manager对象。Employee的克隆方法能够完成这项重任吗？这将取决于Manager类中包含哪些域。在前面列举的示例中，由于bonus域属于基本类型，所以不会出现任何问题。但是，在Manager类中有可能存在一些需要深拷贝的域，或者包含一些没有实现Cloneable接口的域。没有人能够保证子类实现的clone一定正确。鉴于这个原因，应该将Object类中的clone方法声明为protected。但是，如果想让用户调用clone方法，就不能这样做。

在自定义的类中应该实现clone方法吗？如果客户需要深拷贝就应该实现它。有些人认为应该完全避免使用clone，并通过实现其他的方法达到此目的。我们同意这种观点，clone的确显得有点笨拙，但改用其他方法实现这项操作也会遇到同样的问题。至少，克隆的应用并不像人们想像的那样普遍。在标准类库中，只有不到5%的类实现了clone。

在例6-2的程序中，克隆了一个Employee对象，然后，调用了两个改变域值的方法。raiseSalary方法改变了salary域值，setHireDay方法改变了hireDay域值。由于clone实现的是深拷贝，所以对这两个域值的改变并没有影响原始对象。


注释：所有的数组类型均包含一个clone方法，这个方法被设为public，而不是protected。可以利用这个方法创建一个包含所有数据元素拷贝的一个新数组。例如：

```
int[] luckyNumbers = { 2, 3, 5, 7, 11, 13 };
int[] cloned = (int[]) luckyNumbers.clone();
cloned[5] = 12; // doesn't change luckyNumbers[5]
```

✔ **注释**：将在卷II第1章中介绍另一种克隆对象的机制，其中使用了Java的序列化功能。这种机制很容易实现并且也很安全，但效率较低。

例6-2 CloneTest.java

```
import java.util.*;

/**
 * This program demonstrates cloning.
 * @version 1.10 2002-07-01
 * @author Cay Horstmann
 */
public class CloneTest
{
   public static void main(String[] args)
   {
      try
      {
         Employee original = new Employee("John Q. Public", 50000);
         original.setHireDay(2000, 1, 1);
         Employee copy = original.clone();
         copy.raiseSalary(10);
         copy.setHireDay(2002, 12, 31);
         System.out.println("original=" + original);
         System.out.println("copy=" + copy);
      }
      catch (CloneNotSupportedException e)
      {
         e.printStackTrace();
      }
   }
}

class Employee implements Cloneable
{
   public Employee(String n, double s)
   {
      name = n;
      salary = s;
      hireDay = new Date();
   }

   public Employee clone() throws CloneNotSupportedException
   {
      // call Object.clone()
      Employee cloned = (Employee) super.clone();

      // clone mutable fields
      cloned.hireDay = (Date) hireDay.clone();

      return cloned;
   }
```

```

    /**
     * Set the hire day to a given date.
     * @param year the year of the hire day
     * @param month the month of the hire day
     * @param day the day of the hire day
     */
    public void setHireDay(int year, int month, int day)
    {
       Date newHireDay = new GregorianCalendar(year, month - 1, day).getTime();

       // Example of instance field mutation
       hireDay.setTime(newHireDay.getTime());
    }

    public void raiseSalary(double byPercent)
    {
       double raise = salary * byPercent / 100;
       salary += raise;
    }

    public String toString()
    {
       return "Employee[name=" + name + ",salary=" + salary + ",hireDay=" + hireDay + "]";
    }

    private String name;
    private double salary;
    private Date hireDay;
}
```

6.3 接口与回调

回调（callback）是一种常见的程序设计模式。在这种模式中，可以指出某个特定事件发生时应该采取的动作。例如，可以指出在按下鼠标或选择某个菜单项时应该采取什么行动。然而，由于至此还没有介绍如何实现用户接口，所以只能讨论一些与上述操作类似，但比较简单的示例。

在java.swing包中有一个Timer类，可以使用它在到达给定的时间间隔时发出通告。例如，假如程序中有一个时钟，就可以请求每秒钟获得一个通告，以便更新时钟的画面。

在构造定时器时，需要设置一个时间间隔，并告之定时器，当到达时间间隔时需要做些什么操作。

如何告之定时器做什么呢？在很多程序设计语言中，可以提供一个函数名，定时器周期性地调用它。但是，在Java标准类库中的类采用的是面向对象方法。它将某个类的对象传递给定时器，然后，定时器调用这个对象的方法。由于对象可以携带一些附加的信息，所以传递一个对象比传递一个函数要灵活的多。

当然，定时器需要知道调用哪一个方法，并要求传递的对象所属的类实现了java.awt.event包的ActionListener接口。下面是这个接口：

```
public interface ActionListener
```

```
{
   void actionPerformed(ActionEvent event);
}
```

当到达指定的时间间隔时，定时器就调用actionPerformed方法。

C++注释：第5章已经讲过，Java有函数指针的对应物—Method对象。然而，使用起来却比较困难，速度也稍慢一些，并且在编译时不能提供类型的安全性检查。因此，在任何使用C++函数指针的地方，都应该考虑使用Java中的接口。

假设希望每隔10秒种打印一条信息“At the tone, the time is . . .”，然后响一声，就应该定义一个实现ActionListener接口的类，然后将需要执行的语句放在actionPerformed方法中。

```
class TimePrinter implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      Date now = new Date();
      System.out.println("At the tone, the time is " + now);
      Toolkit.getDefaultToolkit().beep();
   }
}
```

需要注意actionPerformed方法的ActionEvent参数。这个参数提供了事件的相关信息，例如，产生这个事件的源对象。有关这方面的详细内容请参看第8章。在这个程序中，事件的详细信息并不重要，因此，可以放心地忽略这个参数。

接下来，构造这个类的一个对象，并将它传递给Timer构造器。

```
ActionListener listener = new TimePrinter();
Timer t = new Timer(10000, listener);
```

Timer构造器的第一个参数是发出通告的时间间隔，它的单位是毫秒。这里希望每隔10秒种通告一次。第二个参数是监听器对象。

最后，启动定时器：

```
t.start();
```

每个10秒钟，下列信息显示一次，然后响一声铃。

```
At the tone, the time is Thu Apr 13 23:29:08 PDT 2000
```

例6-3给出了定时器和监听器的操作行为。在定时器启动以后，程序将弹出一个消息对话框，并等待用户点击Ok按钮来终止程序的执行。在程序等待用户操作的同时，每隔10秒显示一次当前的时间。

运行这个程序时要有一些耐心。程序启动后，将会立即显示一个包含“Quit program?”字样的对话框，10秒钟之后，第1条定时器消息才会显示出来。

需要注意，这个程序除了导入javax.swing.*和java.util.*外，还通过类名导入了javax.swing.Timer。这就消除了javax.swing.Timer与java.util.Timer之间产生的二义性。这里的java.util.Timer是一个与本例无关的类，它主要用于调度后台任务。

例6-3 TimerTest.java

```java
/**
   @version 1.00 2000-04-13
   @author Cay Horstmann
*/

import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;
import javax.swing.Timer;
// to resolve conflict with java.util.Timer

public class TimerTest
{
   public static void main(String[] args)
   {
      ActionListener listener = new TimePrinter();

      // construct a timer that calls the listener
      // once every 10 seconds
      Timer t = new Timer(10000, listener);
      t.start();

      JOptionPane.showMessageDialog(null, "Quit program?");
      System.exit(0);
   }
}

class TimePrinter implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      Date now = new Date();
      System.out.println("At the tone, the time is " + now);
      Toolkit.getDefaultToolkit().beep();
   }
}
```

API javax.swing.JOptionPane 1.2

- `static void showMessageDialog(Component parent, Object message)`

  显示一个包含一条消息和OK按钮的对话框。这个对话框将位于其parent组件的中央。如果parent为null，对话框将显示在屏幕的中央。

API javax.swing.Timer 1.2

- `Timer(int interval, AvtionListener listener)`

  构造一个定时器，每隔interval毫秒钟通告listener一次。

- `void start()`

  启动定时器。一旦启动成功，定时器将调用监听器的actionPerformed。

• void stop()

停止定时器。一旦停止成功，定时器将不再调用监听器的actionPerformed。

API javax.awt.Toolkit 1.0

• static Toolkit getDefaultToolkit()

获得默认的工具箱。工具箱包含有关GUI环境的信息。

• void beep()

发出一声铃响。

6.4 内部类

内部类（inner class）是定义在另一个类中的类。为什么需要使用内部类呢？其主要原因有以下三点：

• 内部类方法可以访问该类定义所在的作用域中的数据，包括私有的数据。
• 内部类可以对同一个包中的其他类隐藏起来。
• 当想要定义一个回调函数且不想编写大量代码时，使用*匿名*（anonymous）内部类比较便捷。

我们将这个比较复杂的内容分几部分介绍。

• 在6.4.1节中，给出一个简单的内部类，它将访问外围类的实例域。
• 在6.4.2节中，给出内部类的特殊语法规则。
• 在6.4.3节中，领略一下内部类的内部，探讨一下如何将其转换成常规类。读者可以跳过这一节。
• 在6.4.4节中，讨论*局部内部类*，它可以访问作用域中的局部变量。
• 在6.4.5节中，介绍*匿名内部类*，说明用于实现回调的基本方法。
• 最后在6.4.6节中，介绍如何将*静态内部类*嵌套在辅助类中。



C++注释：*C++有嵌套类。一个被嵌套的类包含在外围类的作用域内。下面是一个典型的例子，一个链表类定义了一个存储结点的类和一个定义迭代器位置的类。*

```
class LinkedList
{
public:
   class Iterator // a nested class
   {
   public:
      void insert(int x);
      int erase();
      . . .
   };
   . . .
private:
   class Link // a nested class
   {
   public:
      Link* next;
```

```
        int data;
      };
      . . .
   };
```

嵌套是一种类之间的关系，而不是对象之间的关系。一个LinkedList对象并不包含Iterator类型或Link类型的子对象。

嵌套类有两个好处：命名控制和访问控制。由于名字Iterator嵌套在LinkedList类的内部，所以在外部被命名为LinkedList::Iterator，这样就不会与其他名为Iterator的类发生冲突。在Java中这个并不重要，因为Java包已经提供了相同的命名控制。需要注意的是，Link类位于LinkedList类的私有部分，因此，Link对其他的代码均不可见。鉴于此情况，可以将Link的数据域设计为公有的，它仍然是安全的。这些数据域只能被LinkedList类（具有访问这些数据域的合理需要）中的方法访问，而不会暴露给其他的代码。在Java中，只有内部类能够实现这样的控制。

然而，Java内部类还有另外一个功能，这使得它比C++的嵌套类更加丰富，用途更加广泛。内部类的对象有一个隐式引用，它引用了实例化该内部对象的外围类对象。通过这个指针，可以访问外围类对象的全部状态。在本章后续内容中，我们将会看到有关这个Java机制的详细介绍。

在Java中，static内部类没有这种附加指针，这样的内部类与C++中的嵌套类很相似。

6.4.1 使用内部类访问对象状态

内部类的语法比较复杂。鉴于此情况，我们选择一个简单但不太实用的例子说明内部类的使用方式。下面将进一步分析TimerTest示例，并抽象出一个TalkingClock类。构造一个语音时钟时需要提供两个参数：发布通告的间隔和开关铃声的标志。

```
public class TalkingClock
{
   public TalkingClock(int interval, boolean beep) { . . . }
   public void start() { . . . }

   private int interval;
   private boolean beep;

   public class TimePrinter implements ActionListener
      // an inner class
   {
      . . .
   }
}
```

需要注意，这里的TimePrinter类位于TalkingClock类内部。这并不意味着每个TalkingClock都有一个TimePrinter实例域。如前所示，TimePrinter对象是由TalkingClock类的方法构造。

下面是TimePrinter类的详细内容。需要注意一点，actionPerformed方法在发出铃声之前检查了beep标志。

```
private class TimePrinter implements ActionListener
```

```
{
   public void actionPerformed(ActionEvent event)
   {
      Date now = new Date();
      System.out.println("At the tone, the time is " + now);
      if (beep) Toolkit.getDefaultToolkit().beep();
   }
}
```

令人惊讶的事情发生了。TimePrinter类没有实例域或者名为beep的变量，取而代之的是beep引用了创建TimePrinter的TalkingClock对象的域。这是一种创新的想法。从传统意义上讲，一个方法可以引用调用这个方法的对象数据域。内部类既可以访问自身的数据域，也可以访问创建它的外围类对象的数据域。

为了能够运行这个程序，内部类的对象总有一个隐式引用，它指向了创建它的外部类对象。如图6-3所示。

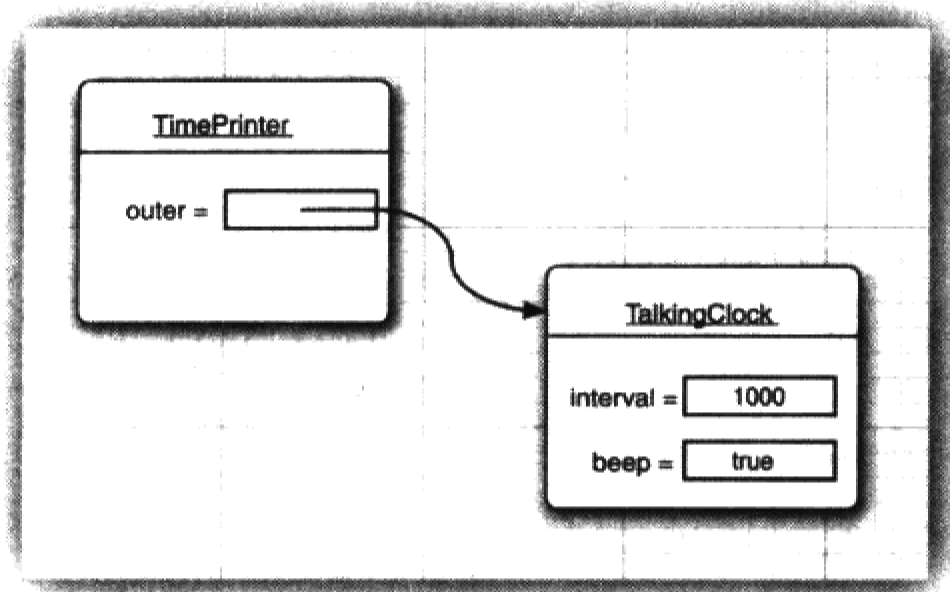


图6-3 内部类对象拥有一个对外围类对象的引用

这个引用在内部类的定义中是不可见的。然而，为了说明这个概念，我们将外围类对象的引用称为outer。于是actionPerformed方法将等价于下列形式：

```
public void actionPerformed(ActionEvent event)
{
   Date now = new Date();
   System.out.println("At the tone, the time is " + now);
   if (outer.beep) Toolkit.getDefaultToolkit().beep();
}
```

外围类的引用在构造器中设置。编译器修改了所有的内部类的构造器，添加一个外围类引用的参数。因为TimePrinter类没有定义构造器，所以编译器为这个类生成了一个默认的构造器，其代码如下所示：

```
public TimePrinter(TalkingClock clock) // automatically generated code
{
   outer = clock;
}
```

请再注意一下，outer不是Java的关键字。我们只是用它说明内部类中的机制。

当在start方法中创建了TimePrinter对象后，编译器就会将this引用传递给当前的语音时钟的构造器：

```
ActionListener listener = new TimePrinter(this); // parameter automatically added
```

例6-4给出了一个测试内部类的完整程序。下面我们再看一下访问控制。如果有一个TimePrinter类是一个常规类，它就需要通过TalkingClock类的公有方法访问beep标志，而使用内部类可以给予改进，即不必提供仅用于访问其他类的访问器。

注释：TimePrinter类被声明为私有的。这样一来，只有TalkingClock的方法才能够生成TimePrinter对象。只有内部类可以是私有类，而常规类只可以具有包可见性，或公有可见性。

例6-4 InnerClassTest.java

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;
import javax.swing.Timer;

/**
 * This program demonstrates the use of inner classes.
 * @version 1.10 2004-02-27
 * @author Cay Horstmann
 */
public class InnerClassTest
{
   public static void main(String[] args)
   {
      TalkingClock clock = new TalkingClock(1000, true);
      clock.start();

      // keep program running until user selects "Ok"
      JOptionPane.showMessageDialog(null, "Quit program?");
      System.exit(0);
   }
}

/**
 * A clock that prints the time in regular intervals.
 */
class TalkingClock
{
   /**
    * Constructs a talking clock
    * @param interval the interval between messages (in milliseconds)
    * @param beep true if the clock should beep
    */
   public TalkingClock(int interval, boolean beep)
   {
      this.interval = interval;
      this.beep = beep;
```

```
   }

   /**
    * Starts the clock.
    */
   public void start()
   {
      ActionListener listener = new TimePrinter();
      Timer t = new Timer(interval, listener);
      t.start();
   }

   private int interval;
   private boolean beep;

   public class TimePrinter implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         Date now = new Date();
         System.out.println("At the tone, the time is " + now);
         if (beep) Toolkit.getDefaultToolkit().beep();
      }
   }
}
```

6.4.2 内部类的特殊语法规则

在上一节中，已经讲述了内部类有一个外围类的引用outer。事实上，使用外围类引用的正规语法还要复杂一些。表达式

OuterClass.this

表示外围类引用。例如，可以像下面这样编写TimePrinter内部类的actionPerformed方法：

```
public void actionPerformed(ActionEvent event)
{
   . . .
   if (TalkingClock.this.beep) Toolkit.getDefaultToolkit().beep();
}
```

反过来，可以采用下列语法格式更加明确地编写内部对象的构造器：

outerObject.new *InnerClass*(*construction parameters*)

例如，

```
ActionListener listener = this.new TimePrinter();
```

在这里，最新构造的TimePrinter对象的外围类引用被设置为创建内部类对象的方法中的this引用。这是一种很常见的情况。通常，this限定词是多余的。不过，可以通过显式地命名将外围类引用设置为其他的对象。例如，如果TimePrinter是一个公有内部类，对于任意的语音时钟都可以构造一个TimePrinter：

```
TalkingClock jabberer = new TalkingClock(1000, true);
TalkingClock.TimePrinter listener = jabberer.new TimePrinter();
```

需要注意，在外围类的作用域之外，可以这样引用内部类：

OuterClass.InnerClass

6.4.3 内部类是否有用、必要和安全

当在Java 1.1的Java语言中增加内部类时，很多程序员都认为这是一项很主要的新特性，但这却违背了Java要比C++更加简单的设计理念。内部类的语法很复杂（可以看到，稍后介绍的匿名内部类更加复杂）。它与访问控制和安全性等其他的语言特性的没有明显的关联。

由于增加了一些看似优美有趣，实属没必要的特性，似乎Java也开始走上了许多语言饱受折磨的毁灭性道路上。

我们并不打算就这个问题给予一个完整的答案。内部类是一种编译器现象，与虚拟机无关。编译器将会把内部类翻译成用$（美元符号）分隔外部类名与内部类名的常规类文件，而虚拟机则对此一无所知。

例如，在TalkingClock类内部的TimePrinter类将被翻译成类文件TalkingClock$TimePrinter.class。为了能够看到执行的效果，可以做一下这个实验：运行第5章中的程序ReflectionTest，并将类TalkingClock$TimePrinter传递给它进行反射。也可以选择简单的使用javap，如下所示：

```
javap -private ClassName
```

注释：*如果使用UNIX，并以命令行的方式提供类名，就需要记住将$字符进行转义。也就是说，应该按照下面这种格式或javap程序运行ReflectionTest程序：*

```
java ReflectionTest TalkingClock\$TimePrinter
```

或

```
javap -private TalkingClock\$TimePrinter
```

这时会看到下面的输出结果：

```
public class TalkingClock$TimePrinter
{
   public TalkingClock$TimePrinter(TalkingClock);

   public void actionPerformed(java.awt.event.ActionEvent);

   final TalkingClock this$0;
}
```

可以清楚地看到，编译器为了引用外围类，生成了一个附加的实例域this$0（名字this$0是由编译器合成的，在自己编写的代码中不能够引用它）。另外，还可以看到构造器的TalkingClock参数。

如果编译器能够自动地进行转换，那么能不能自己编写程序实现这种机制呢？让我们试试看。将TimePrinter定义成一个常规类，并把它置于TalkingClock类的外部。在构造TimePrinter对象的时候，将创建该对象的this指针传递给它。

```
class TalkingClock
{
   . . .
```

```
    public void start()
    {
       ActionListener listener = new TimePrinter(this);
       Timer t = new Timer(interval, listener);
       t.start();
    }
 }

 class TimePrinter implements ActionListener
 {
    public TimePrinter(TalkingClock clock)
    {
        outer = clock;
    }
    . . .
    private TalkingClock outer;
 }
```

现在，看一下actionPerformed方法，它需要访问outer.beep。

```
if (outer.beep) . . . // ERROR
```

这就遇到了一个问题。内部类可以访问外围类的私有数据，但这里的TimePrinter类则不行。

可见，由于内部类拥有访问特权，所以与常规类比较起来功能更加强大。

可能有人会好奇，既然内部类可以被翻译成名字很古怪的常规类（而虚拟机对此一点也不了解），内部类如何管理那些额外的访问特权呢？为了揭开这个谜团，让我们再次利用ReflectTest程序查看一下TalkingClock类：

```
class TalkingClock
{
   public TalkingClock(int, boolean);

   static boolean access$0(TalkingClock);
   public void start();

   private int interval;
   private boolean beep;
}
```

请注意编译器在外围类添加静态方法access$0。它将返回作为参数传递给它的对象域beep。内部类方法将调用那个方法。在TimePrinter类的actionPerformed方法中编写语句：

```
if (beep)
```

将会提高下列调用的效率：

```
if (access$0(outer));
```

这样做不是存在安全风险吗？这种担心是很有道理的。任何人都可以通过调用access$0方法很容易地读取到私有域beep。当然，access$0不是Java的合法方法名。但熟悉类文件结构的黑客可以使用十六进制编辑器轻松地创建一个用虚拟机指令调用那个方法的类文件。由于隐秘地访问方法需要拥有包可见性，所以攻击代码需要与被攻击类放在同一个包中。

总而言之，如果内部类访问了私有数据域，就有可能通过附加在外围类所在包中的其他类访问它们，但做这些事情需要高超的技巧和极大的决心。程序员不可能无意之中就获得对类的

访问权限，而必须刻意地构建或修改类文件才有可能达到这个目的。

✔ **注释**：合成构造器和方法是复杂令人费解的（如果过于注重细节，可以跳过这个注释）。假设将TimePrinter转换为一个内部类。在虚拟机中不存在私有类，因此编译器将会利用私有构造器生成一个包可见的类：

```
private TalkingClock$TimePrinter(TalkingClock);
```

当然，没有人可以调用这个构造器，因此，存在第二个包可见构造器

```
TalkingClock$TimePrinter(TalkingClock, TalkingClock$1);
```

它将调用第一个构造器。

编译器将TalkingClock类start方法中的构造器调用翻译为：

```
new TalkingClock$TimePrinter(this, null)
```

6.4.4 局部内部类

如果仔细地阅读一下TalkingClock示例的代码就会发现，TimePrinter这个类名字只在start方法中创建这个类型的对象时使用了一次。

当遇到这类情况时，可以在一个方法中定义局部类。

```
public void start()
{
   class TimePrinter implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         Date now = new Date();
         System.out.println("At the tone, the time is " + now);
         if (beep) Toolkit.getDefaultToolkit().beep();
      }
   }

   ActionListener listener = new TimePrinter();
   Timer t = new Timer(interval, listener);
   t.start();
}
```

局部类不能用public或private访问说明符进行声明。它的作用域被限定在声明这个局部类的块中。

局部类有一个优势，即对外部世界可以完全地隐藏起来。即使TalkingClock类中的其他代码也不能访问它。除start方法之外，没有任何方法知道TimePrinter类的存在。

6.4.5 由外部方法访问final变量

与其他内部类相比较，局部类还有一个优点。它们不仅能够访问包含它们的外部类，还可以访问局部变量。不过，那些局部变量必须被声明为final。下面是一个典型的示例。这里，将TalkjingClock构造器的参数interval和beep移至start方法中。

```
public void start(int interval, final boolean beep)
{
   class TimePrinter implements ActionListener
```

```
    {
        public void actionPerformed(ActionEvent event)
        {
            Date now = new Date();
            System.out.println("At the tone, the time is " + now);
            if (beep) Toolkit.getDefaultToolkit().beep();
        }
    }

    ActionListener listener = new TimePrinter();
    Timer t = new Timer(interval, listener);
    t.start();
}
```

请注意，TalkingClock类不再需要存储实例变量beep了，它只是引用start方法中的beep参数变量。

这看起来好像没什么值得大惊小怪的。程序行

```
if (beep) . . .
```

毕竟在start方法内部，为什么不能访问beep变量的值呢？

为了能够清楚地看到内部的问题，让我们仔细地考查一下控制流程。

1）调用start方法。

2）调用内部类TimePrinter的构造器，以便初始化对象变量listener。

3）将listener引用传递给Timer构造器，定时器开始计时，start方法结束。此时，start方法的beep参数变量不复存在。

4）然后，actionPerformed方法执行if (beep)...。

为了能够让actionPerformed方法工作，TimePrinter类在beep域释放之前将beep域用start方法的局部变量进行备份。实际上也是这样做的。在我们列举的例子中，编译器为局部内部类构造了名字TalkingClock$TimePrinter。如果再次运行ReflectionTest程序，查看TalkingClock$TimePrinter类，就会看到下列结果：

```
class TalkingClock$1TimePrinter
{
   TalkingClock$1TimePrinter(TalkingClock, boolean);

   public void actionPerformed(java.awt.event.ActionEvent);

   final boolean val$beep;
   final TalkingClock this$0;
}
```

请注意构造器的boolean参数和val$beep实例变量。当创建一个对象的时候，beep就会被传递给构造器，并存储在val$beep域中。编译器必须检测对局部变量的访问，为每一个变量建立相应的数据域，并将局部变量拷贝到构造器中，以便将这些数据域初始化为局部变量的副本。

从程序员的角度看，局部变量的访问非常容易。它减少了需要显式编写的实例域，从而使得内部类更加简单。

前面曾经提到，局部类的方法只可以引用定义为final的局部变量。鉴于此情况，在列举的

示例中，将beep参数声明为final，对它进行初始化后不能够再进行修改。因此，就使得局部变量与在局部类内建立的拷贝保持一致。

注释：前面曾经将final变量作为常量使用，例如：

```
public static final double SPEED_LIMIT = 55;
```

final关键字可以应用于局部变量、实例变量和静态变量。在所有这些情况下，它们的含义都是：在创建这个变量之后，只能够为之赋值一次。此后，再也不能修改它的值了，这就是final。

不过，在定义final变量的时候，不必进行初始化。例如，当调用start方法时，final参数变量beep只能够在创建之后被初始化一次（如果这个方法被调用多次，那么每次调用都有一个新创建的beep参数）。可以看到在Talking$1TimePrinter内部类中的val$beep实例变量仅在内部类的构造器中被设置一次。定义时没有初始化的final变量通常被称为空final（blank final）变量。

有时，final限制显得并不太方便。例如，假设想更新在一个封闭作用域内的计数器。这里想要统计一下在排序过程中调用compareTo方法的次数。

```
int counter = 0;
Date[] dates = new Date[100];
for (int i = 0; i < dates.length; i++)
   dates[i] = new Date()
      {
         public int compareTo(Date other)
         {
            counter++; // ERROR
            return super.compareTo(other);
         }
      };
Arrays.sort(dates);
System.out.println(counter + " comparisons.");
```

由于清楚地知道counter需要更新，所以不能将counter声明为final。由于Integer对象是不可变的，所以也不能用Integer代替它。补救的方法是使用一个长度为1的数组：

```
final int[] counter = new int[1];
for (int i = 0; i < dates.length; i++)
   dates[i] = new Date()
      {
         public int compareTo(Date other)
         {
            counter[0]++;
            return super.compareTo(other);
         }
      };
```

（数组变量仍然被声明为final，但是这仅仅表示不可以让它引用另外一个数组。数组中的数据元素可以自由地更改。）

在内部类被首次提出时，原型编译器对内部类中修改的局部变量自动地进行转换。然而，有些程序员对于编译器在背后生成堆对象感到十分惧怕，而采用final限制代替。在未来的Java

语言版本中有可能会修改这种策略。

6.4.6 匿名内部类

将局部内部类的使用再深入一步。假如只创建这个类的一个对象，就不必命名了。这种类被称为*匿名内部类*（anonymous inner class）。

```
public void start(int interval, final boolean beep)
{
   ActionListener listener = new ActionListener()
      {
         public void actionPerformed(ActionEvent event)
         {
             Date now = new Date();
             System.out.println("At the tone, the time is " + now);
             if (beep) Toolkit.getDefaultToolkit().beep();
         }
      };
   Timer t = new Timer(interval, listener);
   t.start();
}
```

这种语法确实有些难以理解。它的含义是：创建一个实现ActionListener接口的类的新对象，需要实现的方法actionPerformed定义在括号{}内。

用于构造对象的任何参数都要被放在超类名后面的括号( )内。通常的语法格式为：

```
new SuperType(construction parameters)
   {
      inner class methods and data
   }
```

其中，SuperType可以是ActionListener这样的接口，于是内部类就要实现这个接口。SuperType也可以是一个类，于是内部类就要扩展它。

由于构造器的名字必须与类名相同，而匿名类没有类名，所以，匿名类不能有构造器。取而代之的是，将构造器参数传递给*超类*（superclass）构造器。尤其是在内部类实现接口的时候，不能有任何构造参数。不仅如此，还要像下面这样提供一组括号：

```
new InterfaceType()
   {
      methods and data
   }
```

请仔细研究一下，看看构造一个类的新对象与构造一个扩展了那个类的匿名内部类的对象之间有什么差别。

```
Person queen = new Person("Mary");
   // a Person object
Person count = new Person("Dracula") { . . .};
   // an object of an inner class extending Person
```

如果构造参数的闭圆括号跟一个开花括号，正在定义的就是匿名内部类。

匿名内部类是一种好想法呢？还是一种让人迷惑不解的想法呢？也许两者兼有。如果内部类的代码比较短，例如，只有几行简单的代码，匿名内部类就可以节省一些录入的时间。但是

节省这点时间却会让人陷入“混乱的Java代码竞赛”。

例6-5包含了用匿名内部类实现语音时钟程序的全部源代码。将这个程序与例6-4相比较就会发现使用匿名内部类的解决方案比较简短、更切实际、更易于理解。

例6-5 AnonymousInnerClassTest.java

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;
import javax.swing.Timer;

/**
 * This program demonstrates anonymous inner classes.
 * @version 1.10 2004-02-27
 * @author Cay Horstmann
 */
public class AnonymousInnerClassTest
{
   public static void main(String[] args)
   {
      TalkingClock clock = new TalkingClock();
      clock.start(1000, true);

      // keep program running until user selects "Ok"
      JOptionPane.showMessageDialog(null, "Quit program?");
      System.exit(0);
   }
}

/**
 * A clock that prints the time in regular intervals.
 */
class TalkingClock
{
   /**
    * Starts the clock.
    * @param interval the interval between messages (in milliseconds)
    * @param beep true if the clock should beep
    */
   public void start(int interval, final boolean beep)
   {
      ActionListener listener = new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               Date now = new Date();
               System.out.println("At the tone, the time is " + now);
               if (beep) Toolkit.getDefaultToolkit().beep();
            }
         };
      Timer t = new Timer(interval, listener);
      t.start();
   }
}
```

6.4.7 静态内部类

有时候，使用内部类只是为了把一个类隐藏在另外一个类的内部，并不需要内部类引用外围类对象。为此，可以将内部类声明为static，以便取消产生的引用。

下面是一个使用静态内部类的典型例子。考虑一下计算数组中最小值和最大值的问题。当然，可以编写两个方法，一个方法用于计算最小值，另一个方法用于计算最大值。在调用这两个方法的时候，数组被遍历两次。如果只遍历数组一次，并能够同时计算出最小值和最大值，那么就可以大大地提高效率了。

```
double min = Double.MAX_VALUE;
double max = Double.MIN_VALUE;
for (double v : values)
{
   if (min > v) min = v;
   if (max < v) max = v;
}
```

然而，这个方法必须返回两个数值，为此，可以定义一个包含两个值的类pair：

```
class Pair
{
   public Pair(double f, double s)
   {
      first = f;
      second = s;
   }
   public double getFirst() { return first; }
   public double getSecond() {  return second; }

   private double first;
   private double second;
}
```

minmax方法可以返回一个Pair类型的对象。

```
class ArrayAlg
{
   public static Pair minmax(double[] values)
   {
      . . .
      return new Pair(min, max);
   }
}
```

这个方法的调用者可以34使用getFirst和getSecond方法获得答案：

```
Pair p = ArrayAlg.minmax(d);
System.out.println("min = " + p.getFirst());
System.out.println("max = " + p.getSecond());
```

Pair是一个十分大众化的名字。在大型项目中，除了定义包含一对字符串的Pair类之外，其他程序员也很可能使用这个名字。这样就会产生名字冲突。解决这个问题的办法是将Pair定义为ArrayAlg的内部公有类。此后，通过ArrayAlg.Pair访问它：

```
ArrayAlg.Pair p = ArrayAlg.minmax(d);
```

与前面例子中所使用的内部类不同，在Pair对象中不需要引用任何其他的对象，为此，可以将这个内部类声明为static：

```
class ArrayAlg
{
   public static class Pair
   {
      . . .
   }
   . . .
}
```

当然，只有内部类可以声明为static。静态内部类的对象除了没有对生成它的外围类对象的引用特权外，与其他所有内部类完全一样。在我们列举的示例中，必须使用静态内部类，这是由于内部类对象是在静态方法中构造的：

```
public static Pair minmax(double[] d)
{
   . . .
   return new Pair(min, max);
}
```

如果没有将Pair类声明为static，那么编译器将会给出错误报告：没有可用的隐式ArrayAlg类型对象初始化内部类对象。

注释：在内部类不需要访问外围类对象的时候，应该使用静态内部类。有些程序员用嵌套类（nested class）表示静态内部类。

注释：声明在接口中的内部类自动成为static和public。

例6-6包含ArrayAlg类和嵌套的Pair类的全部源代码。

例6-6 StaticInnerClassTest.java

```
/**
 * This program demonstrates the use of static inner classes.
 * @version 1.01 2004-02-27
 * @author Cay Horstmann
 */
public class StaticInnerClassTest
{
   public static void main(String[] args)
   {
      double[] d = new double[20];
      for (int i = 0; i < d.length; i++)
         d[i] = 100 * Math.random();
      ArrayAlg.Pair p = ArrayAlg.minmax(d);
      System.out.println("min = " + p.getFirst());
      System.out.println("max = " + p.getSecond());
   }
}

class ArrayAlg
{
```

```
   /**
    * A pair of floating-point numbers
    */
   public static class Pair
   {
      /**
       * Constructs a pair from two floating-point numbers
       * @param f the first number
       * @param s the second number
       */
      public Pair(double f, double s)
      {
         first = f;
         second = s;
      }

      /**
       * Returns the first number of the pair
       * @return the first number
       */
      public double getFirst()
      {
         return first;
      }

      /**
       * Returns the second number of the pair
       * @return the second number
       */
      public double getSecond()
      {
         return second;
      }

      private double first;
      private double second;
   }

   /**
    * Computes both the minimum and the maximum of an array
    * @param values an array of floating-point numbers
    * @return a pair whose first element is the minimum and whose second element
    * is the maximum
    */
   public static Pair minmax(double[] values)
   {
      double min = Double.MAX_VALUE;
      double max = Double.MIN_VALUE;
      for (double v : values)
      {
         if (min > v) min = v;
         if (max < v) max = v;
      }
      return new Pair(min, max);
   }
}
```

6.5 代理

在本章的最后，讨论一下代理（proxy），这是Java SE 1.3新增加的特性。利用代理可以在运行时创建一个实现了一组给定接口的新类。这种功能只有在编译时无法确定需要实现哪个接口时才有必要使用。对于应用程序设计人员来说，遇到这种情况的机会很少。如果对这种高级技术不感兴趣，可以跳过本节内容。然而，对于系统程序设计人员来说，代理带来的灵活性却十分重要。

假设有一个表示接口的Class对象（有可能只包含一个接口），它的确切类型在编译时无法知道。这确实有些难度。要想构造一个实现这些接口的类，就需要使用newInstance方法或反射找出这个类的构造器。但是，不能实例化一个接口，需要在程序处于运行状态时定义一个新类。

为了解决这个问题，有些程序将会生成代码；将这些代码放置在一个文件中；调用编译器；然后再加载结果类文件。很自然，这样做的速度会比较慢，并且需要将编译器与程序放在一起。而代理机制则是一种更好的解决方案。代理类可以在运行时创建全新的类。这样的代理类能够实现指定的接口。尤其是，它具有下列方法：

- 指定接口所需要的全部方法。
- Object类中的全部方法，例如，toString、equals等。

然而，不能在运行时定义这些方法的新代码。而是要提供一个*调用处理器*（invocation handler）。调用处理器是实现了InvocationHandler接口的类对象。在这个接口中只有一个方法：

```
Object invoke(Object proxy, Method method, Object[] args)
```

无论何时调用代理对象的方法，调用处理器的invoke方法都会被调用，并向其传递Method对象和原始的调用参数。调用处理器必须给出处理调用的方式。

要想创建一个代理对象，需要使用Proxy类的newProxyInstance方法。这个方法有三个参数：

- 一个*类加载器*（class loader）。作为Java安全模型的一部分，对于系统类和从因特网上下载下来的类，可以使用不同的类加载器。有关类加载器的详细内容将在卷II第9章中讨论。目前，用null表示使用默认的类加载器。
- 一个Class对象数组，每个元素都是需要实现的接口。
- 一个调用处理器。

还有两个需要解决的问题。如何定义一个处理器？能够用结果代理对象做些什么？当然，这两个问题的答案取决于打算使用代理机制解决什么问题。使用代理可能出于很多原因，例如：

- 路由对远程服务器的方法调用。
- 在程序运行期间，将用户接口事件与动作关联起来。
- 为调试，跟踪方法调用。

在列举的示例中，使用代理和调用处理器跟踪方法调用，并且定义了一个TraceHander包装器类存储包装的对象。其中的invoke方法打印出被调用方法的名字和参数，随后用包装好的对象作为隐式参数调用这个方法。

```
class TraceHandler implements InvocationHandler
{
   public TraceHandler(Object t)
```

```
   {
      target = t;
   }

   public Object invoke(Object proxy, Method m, Object[] args)
      throws Throwable
   {
      // print method name and parameters
      . . .
      // invoke actual method
      return m.invoke(target, args);
   }

   private Object target;
}
```

下面说明一下如何构造用于跟踪方法调用的代理对象。

```
Object value = . . .;
// construct wrapper
InvocationHandler handler = new TraceHandler(value);
// construct proxy for one or more interfaces
Class[] interfaces = new Class[] { Comparable.class };
Object proxy = Proxy.newProxyInstance(null, interfaces, handler);
```

现在，无论何时用proxy调用某个方法，这个方法的名字和参数就会打印出来，之后再用value调用它。

在例6-7给出的程序中，使用代理对象对二分查找进行跟踪。这里，首先将用1～1000整数的代理填充数组，然后调用Arrays类中的binarySearch方法在数组中查找一个随机整数。最后，打印出与之匹配的元素。

```
Object[] elements = new Object[1000];
// fill elements with proxies for the integers 1 . . . 1000
for (int i = 0; i < elements.length; i++)
{
   Integer value = i + 1;
   elements[i] = Proxy.newInstance(. . .); // proxy for value;
}

// construct a random integer
Integer key = new Random().nextInt(elements.length) + 1;

// search for the key
int result = Arrays.binarySearch(elements, key);

// print match if found
if (result >= 0) System.out.println(elements[result]);
```

在上述代码中，Integer类实现了Comparable接口。代理对象属于在运行时定义的类（它有一个名字，如$Proxy0）。这个类也实现了Comparable接口。然而，它的compareTo方法调用了代理对象处理器的invoke方法。



注释：前面已经讲过，在Java SE 5.0中，Integer类实际上实现了Comparable<Integer>。然而，在运行时，所有的泛型类都被取消，代理将它们构造为原Comparable类的类对象。

binarySearch方法按下面这种方式调用：

```
if (elements[i].compareTo(key) < 0) . . .
```

由于数组中填充了代理对象，所以compareTo调用了TraceHander类中的invoke方法。这个方法打印出了方法名和参数，之后用包装好的Integer对象调用compareTo。

最后，在示例程序的结尾调用：

```
System.out.println(elements[result]);
```

println方法调用代理对象的toString，这个调用也会被重定向到调用处理器上。下面是程序运行的全部跟踪结果：

```
500.compareTo(288)
250.compareTo(288)
375.compareTo(288)
312.compareTo(288)
281.compareTo(288)
296.compareTo(288)
288.compareTo(288)
288.toString()
```

可以看出，二分查找算法查找关键字的过程，即每一步都将查找区间缩减一半。注意，即使不属于Comparable接口，toString方法也被代理。在下一节中会看到，有相当一部分的Object方法都被代理。

例6-7 ProxyTest.java

```
import java.lang.reflect.*;
import java.util.*;

/**
 * This program demonstrates the use of proxies.
 * @version 1.00 2000-04-13
 * @author Cay Horstmann
 */
public class ProxyTest
{
   public static void main(String[] args)
   {
      Object[] elements = new Object[1000];

      // fill elements with proxies for the integers 1 ... 1000
      for (int i = 0; i < elements.length; i++)
      {
         Integer value = i + 1;
         InvocationHandler handler = new TraceHandler(value);
         Object proxy = Proxy.newProxyInstance(null, new Class[] { Comparable.class } , handler);
         elements[i] = proxy;
      }

      // construct a random integer
      Integer key = new Random().nextInt(elements.length) + 1;

      // search for the key
      int result = Arrays.binarySearch(elements, key);
```

```
      // print match if found
      if (result >= 0) System.out.println(elements[result]);
   }
}

/**
 * An invocation handler that prints out the method name and parameters, then
 * invokes the original method
 */
class TraceHandler implements InvocationHandler
{
   /**
    * Constructs a TraceHandler
    * @param t the implicit parameter of the method call
    */
   public TraceHandler(Object t)
   {
      target = t;
   }

   public Object invoke(Object proxy, Method m, Object[] args) throws Throwable
   {
      // print implicit argument
      System.out.print(target);
      // print method name
      System.out.print("." + m.getName() + "(");
      // print explicit arguments
      if (args != null)
      {
         for (int i = 0; i < args.length; i++)
         {
            System.out.print(args[i]);
            if (i < args.length - 1) System.out.print(", ");
         }
      }
      System.out.println(")");

      // invoke actual method
      return m.invoke(target, args);
   }

   private Object target;
}
```

代理类的特性

现在，我们已经看到了代理类的应用，接下来了解它们的一些特性。需要记住，代理类是在程序运行过程中创建的。然而，一旦被创建，就变成了常规类，与虚拟机中的任何其他类没有什么区别。

所有的代理类都扩展于Proxy类。一个代理类只有一个实例域——调用处理器，它定义在Proxy的超类中。为了履行代理对象的职责，所需要的任何附加数据都必须存储在调用处理器中。例如，在例6-7给出的程序中，代理Comparable对象时，TraceHandler包装了实际的对象。

所有的代理类都覆盖了Object类中的方法toString、equals和hashCode。如同所有的代理方法一样，这些方法仅仅调用了调用处理器的invoke。Object类中的其他方法（如clone和getClass）没有被重新定义。

没有定义代理类的名字，Sun虚拟机中的Proxy类将生成一个以字符串$Proxy开头的类名。

对于特定的类加载器和预设的一组接口来说，只能有一个代理类。也就是说，如果使用同一个类加载器和接口数组调用两次newProxyInstance方法的话，那么只能够得到同一个类的两个对象，也可以利用getProxyClass方法获得这个类：

```
Class proxyClass = Proxy.getProxyClass(null, interfaces);
```

代理类一定是public和final。如果代理类实现的所有接口都是public，代理类就不属于某个特定的包；否则，所有非公有的接口都必须属于同一个包，同时，代理类也属于这个包。

可以通过调用Proxy类中的isProxyClass方法检测一个特定的Class对象是否代表一个代理类。

API `java.lang.reflect.InvocationHandler` 1.3

- `Object invoke(Object proxy,Method method,Object[] args)`

  定义了代理对象调用方法时希望执行的动作。

API `java.lang.reflect.Proxy` 1.3

- `static Class getProxyClass(ClassLoader loader,Class[] interfaces)`

  返回实现指定接口的代理类。
- `static Object newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler handler)`

  构造一个实现指定接口的代理类的实例。所有方法都将调用给定处理器对象的invoke方法。
- `static boolean isProxyClass(Class c)`

  如果c是一个代理类返回true。

到此为止，Java程序设计语言的基础概念介绍完毕了。接口和内部类是两个经常使用的概念。然而，前面已经提到过，代理是一项工具构造者感兴趣的高级技术，对应用程序员来说，并不十分重要。下面准备继续学习由第7章开始介绍的图形和用户接口。

第7章 图形程序设计

- ▲ Swing概述
- ▲ 创建框架
- ▲ 框架定位
- ▲ 在组件中显示信息
- ▲ 2D图形
- ▲ 颜色
- ▲ 字体
- ▲ 图像

到目前为止，我们编写的程序都是通过键盘接收输入，在控制台屏幕上显示结果。绝大多数用户并不喜欢这种交互方式。现代的程序早已不采用这种操作方法了，网络程序更是如此。从本章开始，将介绍如何编写使用图形用户界面（GUI）的Java程序。本章主要讲述如何编写定义屏幕上的窗口大小和位置的程序；如何在窗口中采用多种字体显示文本；如何显示图像等。这些都是需要掌握的编程技能，在后续各章中，将会使用这些技术编写一些很有趣味的程序。

随后的两章，将介绍如何处理敲击键盘，点击鼠标这样的事件，以及如何在应用程序中添加菜单和按钮这样的界面元素。学习完这三章之后，读者就应该能够掌握编写独立运行的图形应用程序的要素了。有关更加复杂的图形程序设计技巧请参看卷II。

另外，如果只希望用Java编写服务器端的应用程序，并且对GUI 编程又不感兴趣，那么就可以跳过这几个章节。

7.1 Swing概述

在Java 1.0刚刚出现的时候，包含了一个用于基本GUI程序设计的类库，Sun将它称为抽象窗口工具箱（Abstract Window Toolkit，AWT）。基本AWT库采用将处理用户界面元素的任务委派给每个目标平台（Windows、Solaris、Macintosh等）的本地GUI工具箱的方式，由本地GUI工具箱负责用户界面元素的创建和动作。例如，如果使用最初的AWT在Java窗口中放置一个文本框，就会有一个低层的“对等体”文本框，用来真正地处理文本输入。从理论上说，结果程序可以运行在任何平台上，但观感（look and feel）的效果却依赖于目标平台，因此，Sun公司的口号是“一次编写，随处使用”。

对于简单的应用程序来说，基于对等体方法的效果还是不错的。但是，要想编写依赖于本地用户界面元素的高质量、可移植的图形库就会暴露出一些缺陷。例如，菜单、滚动条和文本域这些用户界面元素，在不同的平台上，操作行为存在着一些微妙的差别。因此，要想给予用户一致的、可预见性的界面操作方式是相当困难的。而且，有些图形环境（如X11/Motif）并没有像Windows或Macintosh这样丰富的用户界面组件集合。这也就将基于对等体的可移植库限制在了“最小公分母”的范围内。其结果使AWT构建的GUI应用程序看起来没有Windows或Macintosh应用程序显示的那么漂亮，也没有提供那些平台用户所认知的功能。更加糟糕的是，

在不同平台上的AWT用户界面库中存在着不同的bug。研发人员必须在每一个平台上测试应用程序，因此人们嘲弄地将AWT称为“一次编写，随处调试”。

在1996年，Netscape创建了一种称为IFC（Internet Foundation Class）的GUI库，它采用了与AWT完全不同的工作方式。它将按钮、菜单这样的用户界面元素绘制在空白窗口上，而对等体只需要创建和绘制窗口。因此，Netscape的IFC组件在程序运行的所有平台上的外观和动作都一样。Sun与Netscape合作完善了这种方式，创建了一个名为Swing的用户界面库。Swing可作为Java 1.1的扩展部分使用，现已成为Java SE 1.2标准库的一部分。

就像Duke Ellington所说的那样：“如果没有Swing，Java图形界面就没有任何意义”。现在，Swing是不对等基于GUI工具箱的正式名字。它已是Java基础类库（Java Foundation Class，JFC）的一部分。完整的JFC十分庞大，其中包含的内容远远大于Swing GUI工具箱。JFC特性不仅仅包含了Swing组件，而且还包含了一个可访问的API、一个2D API和一个可拖拽的API。

注释：Swing没有完全替代AWT，而是基于AWT架构之上。Swing仅仅提供了能力更加强大的用户界面组件。尤其在采用Swing编写的程序中，还需要使用基本的AWT处理事件。从现在开始，Swing是指“被绘制的”用户界面类；AWT是指像事件处理这样的窗口工具箱的低层机制。

当然，在用户屏幕上显示基于Swing用户界面的元素要比显示AWT的基于对等体组件的速度慢一些。鉴于以往的经验，对于任何一台现代的计算机来说，微小的速度差别无妨大碍。另外，由于下列几点无法抗拒的原因，人们选择Swing：

- Swing拥有一个丰富、便捷的用户界面元素集合。
- Swing对低层平台依赖的很少，因此与平台相关的bug很少。
- Swing给予不同平台的用户一致的感觉。

不过，上面第三点存在着一个潜在的问题：如果在所有平台上用户界面元素看起来都一样，那么它们就有可能与本地控件不一样，而这些平台的用户对此可能并不熟悉。

Swing采用了一种很巧妙的方式来解决这个问题。在程序员编写Swing程序时，可以为程序指定专门的“观感”。例如，图7-1和图7-2展示了同一个程序在Windows和Motif平台下运行的观感。

此外，Sun开发了一种称为Metal的独立于平台的观感。现在，市场上人们将它称为“Java观感”。不过，绝大多数程序员还继续沿用Metal这个术语，在本书中也将这样称呼。

有些人批评Metal有点笨重，而在Java SE 5.0中看起来却焕然一新（参见图7-3）。现在，Metal外观支持多种主题，每一种主题的颜色和字体都有微小的变化。默认的主题叫做Ocean。

在Java SE 6中，Sun改进了对Windows和GTK本地观感的支持。Swing应用程序将会支持色彩主题的个性化设置，逼真地表现着动态按钮和变得十分时尚的滚动条。

有些用户更希望Java应用程序采用本地平台的观感，而不采用Metal或第三方的观感。在第8章中你将会看到，让用户随意地选择喜欢的观感是非常容易的。

注释：虽然在这本书中没有用很多篇幅介绍有关设定“观感”的方式，但Java程序员能够扩展一个已存在的观感，甚至设计一个全新的观感。设计Swing组件的绘制方式是一个很

繁琐的过程。有些程序员已经尝试过一些这样的工作，尤其是将Java移植到信息亭终端和手持设备这样的非传统平台上。有关一些有趣的观感的实现请参看http://www.javootoo.com。Java SE 5.0引入了一种被称为Synth的新观感方式，使用它处理比较容易。在Synth中，可以通过提供图像文件和XML描述符来定义一种新的观感，而不需要编写任何代码。

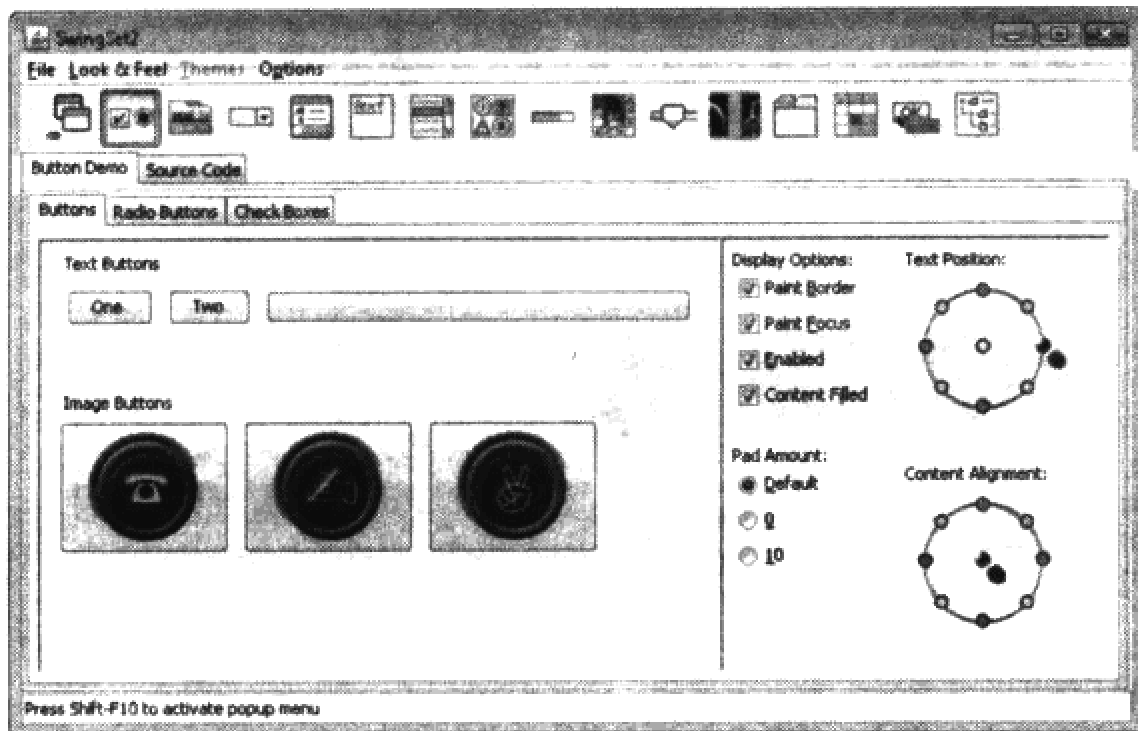


图7-1 Swing的Window观感

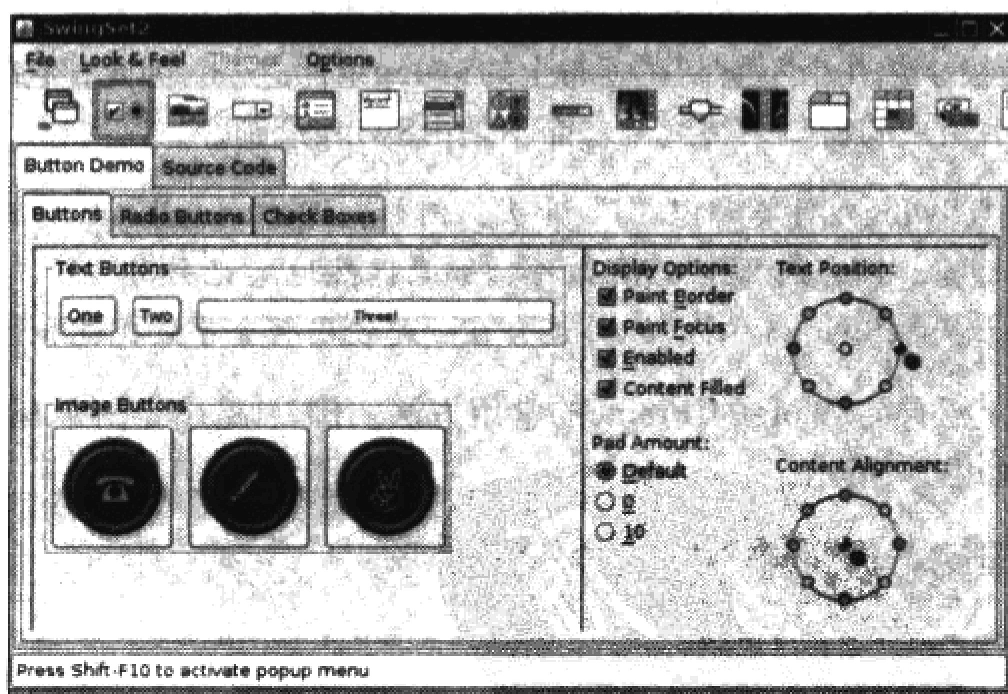


图7-2 Swing的GTK观感

注释：绝大多数Java用户界面程序设计都采用Swing，但有一个特别的例外。Eclipse集成开发环境使用了一种与AWT类似，且被称为SWT的图形工具箱，它可以映射到不同平台的本地组件上。有关SWT的描述可以在网站http://www.eclipse.org/ articles/找到。

最后，给大家一个忠告，如果使用过Visual Baisc或C#编写Microsoft Windows应用程序，就应该了解这些产品提供的图形布局工具和资源编辑器带来的便利。这些工具可以用来设计应

用程序的外观，然后生成大部分（有时是全部）GUI代码。尽管也有一些Java程序设计的GUI构造器，但要想有效地使用这些工具，需要知道如何手工地创建用户界面。本章剩余的部分将介绍关于显示一个窗口及绘制内容的基本知识。

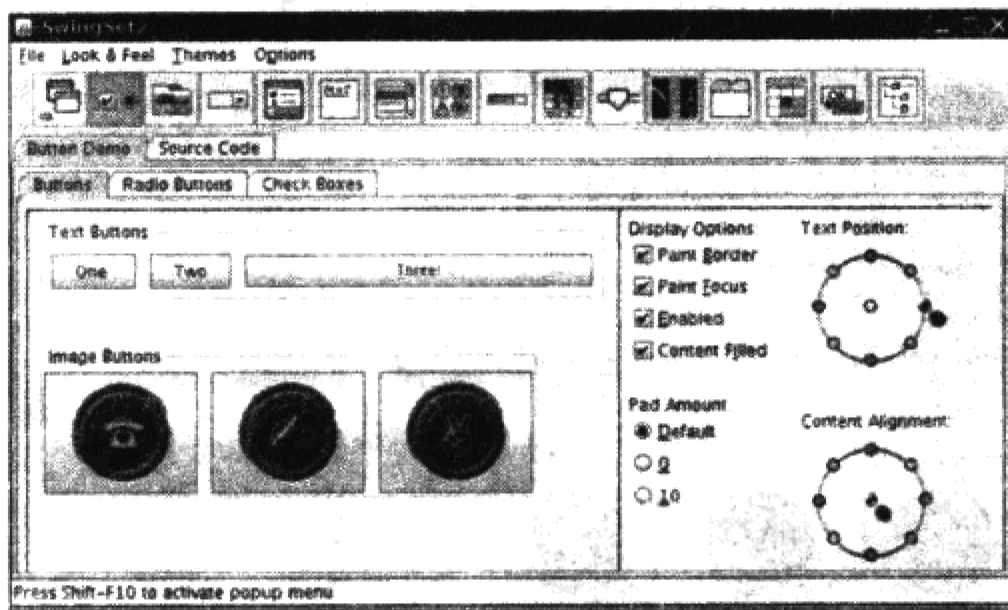


图7-3 Metal观感的Ocean主题

7.2 创建框架

在Java中，顶层窗口（就是没有包含在其他窗口中的窗口）被称为框架（frame）。在AWT库中有一个称为Frame的类，用于描述顶层窗口。这个类的Swing版本名为JFrame，它扩展于Frame类。JFrame是极少数几个不绘制在画布上的Swing组件之一。因此，它的修饰部件（按钮、标题栏、图标等）由用户的窗口系统绘制，而不是由Swing绘制。

警告：绝大多数Swing组件类都以“J”开头，例如，JButton、JFrame等。在Java中有Button和Frame这样的类，但它们属于AWT组件。如果偶然地忘记书写“J”，程序仍然可以进行编译和运行，但是将Swing和AWT组件混合在一起使用将会导致视觉和行为的不一致。

在本节中，将介绍有关Swing的JFrame的常用方法。例7-1给出了一个在屏幕中显示一个空框架的简单程序，如图7-4所示。

例7-1 SimpleFrameTest.java

```
import javax.swing.*;

/**
 * @version 1.32 2007-06-12
 * @author Cay Horstmann
 */
public class SimpleFrameTest
{
   public static void main(String[] args)
   {
      SimpleFrame frame = new SimpleFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
```

```
    }
}

class SimpleFrame extends JFrame
{
    public SimpleFrame()
    {
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
    }

    public static final int DEFAULT_WIDTH = 300;
    public static final int DEFAULT_HEIGHT = 200;
}
```

下面逐行地讲解一下这个程序。

Swing类位于javax.swing包中。包名javax表示这是一个Java扩展包，而不是核心包。Swing类实际上是对Java 1.1的扩展。从1.2版本开始，在每个Java SE实现中都包含它。

在默认情况下，框架的大小为0×0，这种框架没有什么实际意义。这里定义了一个子类SimpleFrame，它的构造器将框架大小设置为300×200像素。这是SimpleFrame和JFrame之间惟一的差别。

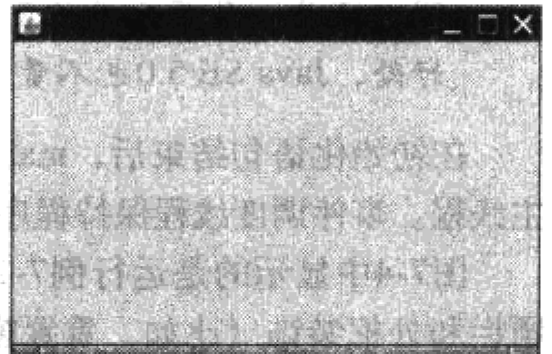


图7-4 最简单的可见框架

在SimpleFrameTest类的main方法中，由创建一个SmpleFrame对象开始程序的运行。

在每个Swing程序中，有两点技术需要强调。

首先，所有的Swing组件必须由*事件调度线程*（event dispatch thread）进行配置，线程将鼠标点击和键盘敲击控制转移到用户接口组件。下面的代码片断是事件调度线程中的执行代码：

```
EventQueue.invokeLater(new Runnable()
   {
      public void run()
      {
         statements
      }
   });
```

这一内容将在14章中详细讨论。现在，只需要简单地将其看作启动一个Swing程序的神奇代码。



注释：许多Swing程序并没有在事件调度线程中初始化用户接口。在主线程中完成初始化是通常采用的方式。遗憾的是，由于Swing组件十分复杂，Sun的程序员无法保证这种方式的安全性。虽然发生错误的概率非常小，但任何人不愿意成为遭遇这个问题的不幸者之一。即使代码看起来有些神秘，也最好能够保证其正确性。

接下来，定义一个用户关闭这个框架时的响应动作。对于这个程序而言，只让程序简单地退出即可。选择这个响应动作的语句是

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

在包含多个框架的程序中，不能在用户关闭其中的一个框架时就让程序退出。在默认情况

下，用户关闭窗口时只是将框架隐藏起来，而程序并没有终止（在最后一个框架不可见之后，程序再终止，这样处理比较合适，而Swing却不是这样工作的）。

简单地构造框架是不会自动地显示出来的，框架起初是不可见的。这就给程序员了一个机会，可以在框架第一次显示之前往其中添加组件。为了显示框架，main方法需要调用框架的setVisible方法。

注释：在Java SE 5.0以前的版本中，可以使用JFrame类从超类Window继承show方法。Window类的超类是Component，其中也有一个show方法。在Java SE 1.2中不提倡使用Component.show。如果想要显示一个组件，建议调用setVisible(true)。然而，Java SE 1.4以前的版本，并没有反对使用Window.show方法。事实上，这个方法很实用，它可以让窗口可见，且置于其他窗口的前面。遗憾的是，由于不提倡使用它，随之也失去了这一好处，Java SE 5.0也不赞成使用show显示窗口。

在初始化语句结束后，main方法退出。需要注意，退出main并没有终止程序，终止的只是主线程。事件调度线程保持程序处于激活状态，直到关闭框架或调用System.exit方法终止程序。

图7-4中显示的是运行例7-1的结果，它只是一个很枯燥的顶层窗口。在这个图中看到的标题栏和外框装饰（比如，重置窗口大小的拐角）都是由操作系统绘制的，而不是Swing库。在Windows，GTK或Mac下运行同样的程序，将会得到不同的框架装饰。Swing库负责绘制框架内的所有内容。在这个程序中，只用默认的背景色填充了框架。

注释：在Java SE 1.4中，可以调用frame.setUndecorated(true) 关闭所有框架装饰。

7.3 框架定位

JFrame类本身只包含若干个改变框架外观的方法。然而，通过继承从JFrame的各个超类中继承了许多用于处理框架大小和位置的方法。其中最重要的有下面几个：

- setLocation和setBounds方法用于设置框架的位置。
- setIconImage用于告诉窗口系统在标题栏、任务切换窗口等位置显示哪个图标。
- setTitle用于改变标题栏的文字。
- setResizable利用一个boolean值确定框架的大小是否允许用户改变。

图7-5给出了JFrame类的继承层次。

提示：本节API注解给出了一些非常重要的用于设置框架适当观感的方法。其中一些定义在JFrame类中，而另一些来自于JFrame的各个超类。因此，可能需要查阅API文档，以便确定是否存在能够实现某个特定目的的方法。遗憾的是，在文档中查阅一个类继承的方法是一件比较令人烦恼的事情。对于子类来说，API文档只解释了覆盖的方法。例如，可以应用于JFrame类对象的toFront方法，由于它是从Window类继承而来的，所以JFrame文档没有对它进行解释。如果认为应该有一个能够完成某项操作的方法，而在处理的类文档中又没有解释，就应该查看这个类的超类API文档。每页API上面都有一个对超类的超链接，继承方法被列在新方法和覆盖方法的汇总下面。

正像API注解中所说的，对Component类（是所有GUI对象的祖先）和Window类（是Frame类的超类）需要仔细地研究一下，从中找到缩放和改变框架的方法。例如，在Component类中的setLocation方法是重定位组件的一个方法。如果调用

```
setLocation(x, y)
```

则窗口将放置在左上角水平x像素，垂直y像素的位置，坐标（0，0）位于屏幕的左上角。同样地，Component中的setBounds方法可以实现一步重定位组件（特别是JFrame）大小和位置的操作，例如：

```
setBounds(x, y, width, height)
```

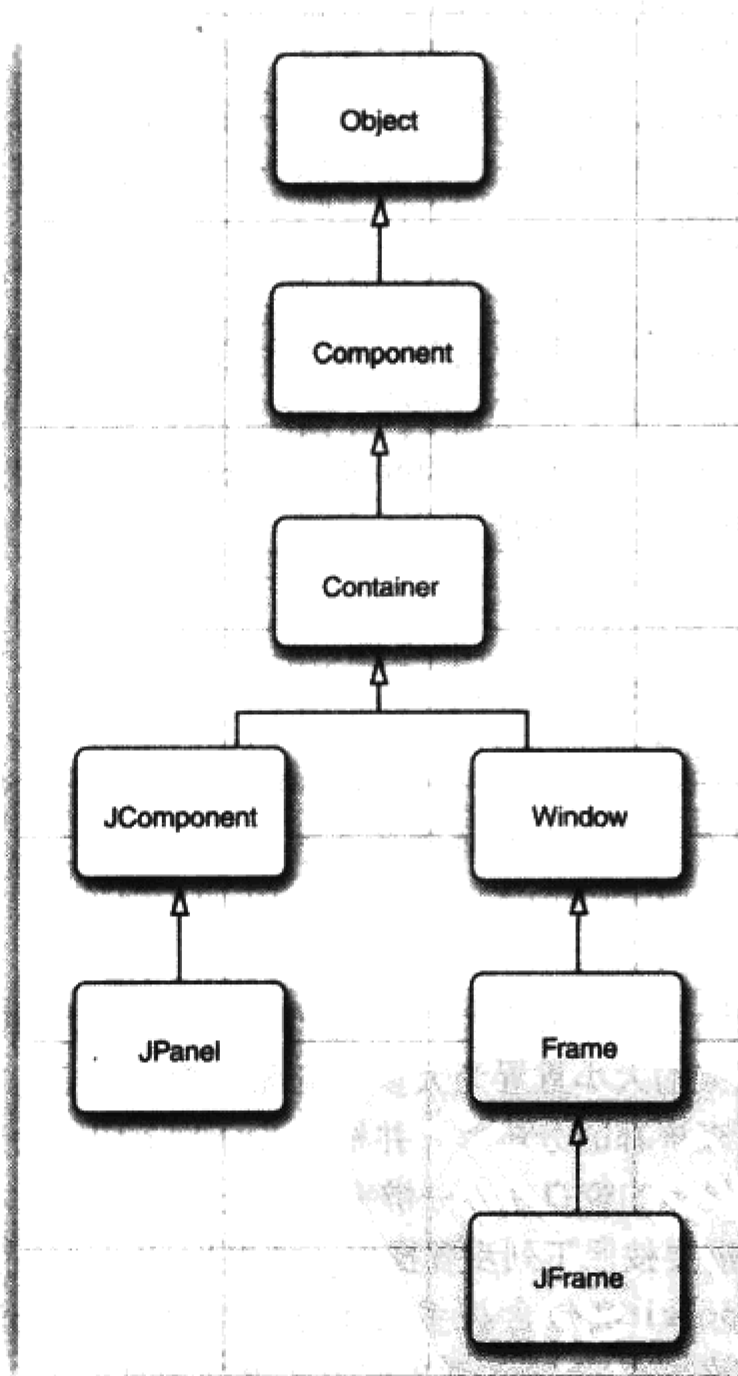


图7-5 AWT和Swing中框架和组件类的继承层次

可以让窗口系统控制窗口的位置，如果在显示窗口之前调用

```
setLoationByPlatform(true);
```

窗口系统会拾取窗口的位置（而不是大小），通常是距最后一个显示窗口很少偏移量的位置。

注释：对于框架来说，setLocation和setBounds中的坐标均相对于整个屏幕。在第9章中将会看到，在容器中包含的组件所指的坐标均相对于容器。

7.4 框架属性

组件类的很多方法是以获取/设置这一对操作形式出现的，例如，Frame类的下列方法：

```
public String getTitle()
public void setTitle(String title)
```

这样的一个获取/设置对被称为一种*属性*。属性包含属性名和类型。将get或set之后的第一个字母改为小写字母就可以得到相应的属性名。例如，Frame类有一个名为title且类型为String的属性。

从概念上讲，title是框架的一个属性。当设置这个属性时，希望这个标题能够改变用户屏幕上的显示。当获取这个属性时，希望能够返回已经设置的属性值。

我们并不清楚（也不关心）Frame类是如何实现这个属性的。或许只是简单的利用对等框架存储标题。或许有一个实例域：

```
private String title; // not required for property
```

如果类没有匹配的实例域，我们将不清楚（也不关心）如何实现获取和设置方法。或许只是读、写实例域，或许还执行了很多其他的操作。例如，当标题发生变化时，通知给窗口系统。

针对get/set约定有一个例外：对于类型为boolean的属性，获取类方法由is开头。例如，下面两个方法定义了locationByPlatform属性：

```
public boolean isLocationByPlatform()
public void setLocationByPlatform(boolean b)
```

有关属性的详细内容，请参看卷II第8章。

注释：许多程序设计语言（特别是，Visual Basic和C#）已经内置了对属性的支持。在Java未来的版本中，也有可能提供对属性的支持。

7.5 决定框架大小

请注意：如果没有明确地指定框架的大小，所有框架的默认值为0×0像素。为了让示例程序尽可能地简单，这里将框架的大小重置为大多数情况下都可以接受的显示尺寸。然而，对于专业应用程序来说，应该检查屏幕的分辨率，并根据其分辨率编写代码重置框架的大小，如在膝上型电脑的屏幕上，正常显示的窗口在高分辨率屏幕上可能会变成一张邮票的大小。

为了得到屏幕的大小，需要按照下列步骤操作。调用Toolkit类的静态方法getDefaultToolkit得到一个Toolkit对象（Toolkit类包含很多与本地窗口系统打交道的方法）。然后，调用getScreenSize方法，这个方法以Dimension对象的形式返回屏幕的大小。Dimension对象同时用公有实例变量width和height保存着屏幕的宽度和高度。下面是相关的代码：

```
Toolkit kit = Toolkit.getDefaultToolkit();
Dimension screenSize = kit.getScreenSize();
int screenWidth = screenSize.width;
int screenHeight = screenSize.height;
```

下面，将框架大小设定为上面取值的50%，然后，告知窗口系统定位框架：

```
setSize(screenWidth / 2, screenHeight / 2);
setLocationByPlatform(true);
```

另外，还提供一个图标。由于图像的描述与系统有关，所以需要再次使用工具箱加载图像。然后，将这个图像设置为框架的图标。

```
Image img = kit.getImage("icon.gif");
setIconImage(img);
```

对于不同的操作系统，所看到的图标显示位置有可能不同。例如，在Windows中，图标显示在窗口的左上角，按下ALT+TAB，可以在活动任务的列表中看到相应程序的图标。

例7-2是完整的程序。当运行程序时，请注意看“Core Java”图标。

下面是为了处理框架给予的一些提示：

- 如果框架中只包含标准的组件，如按钮和文本框，那么可以通过调用pack方法设置框架大小。框架将被设置为刚好能够放置所有组件的大小。在通常情况下，将程序的主框架尺寸设置为最大。正如Java SE 1.4，可以通过调用下列方法将框架设置为最大。

  ```
  frame.setExtendedState(Frame.MAXIMIZED_BOTH);
  ```

- 牢记用户定位应用程序的框架位置、重置框架大小，并且在应用程序再次启动时恢复这些内容是一个不错的想法。在第10章中将会介绍如何运用API的参数选择达到这个目的。
- 如果编写一个使用多个显示屏幕的应用程序，应该利用GraphicsEnvironment和GraphicsDevice类获得显示屏幕的大小。
- GraphicsDevice类允许在全屏模式下执行应用程序。

例7-2 SizedFrameTest.java

```
import java.awt.*;

import javax.swing.*;

/**
 * @version 1.32 2007-04-14
 * @author Cay Horstmann
 */
public class SizedFrameTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               SizedFrame frame = new SizedFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

class SizedFrame extends JFrame
```

```
{
   public SizedFrame()
   {
      // get screen dimensions

      Toolkit kit = Toolkit.getDefaultToolkit();
      Dimension screenSize = kit.getScreenSize();
      int screenHeight = screenSize.height;
      int screenWidth = screenSize.width;

      // set frame width, height and let platform pick screen location

      setSize(screenWidth / 2, screenHeight / 2);
      setLocationByPlatform(true);

      // set frame icon and title

      Image img = kit.getImage("icon.gif");
      setIconImage(img);
      setTitle("SizedFrame");
   }
}
```

API java.awt.Component 1.0

- `boolean isVisible()`
- `void setVisible(boolean b)`

  获取或设置visible属性。组件最初是可见的，但JFrame这样的顶层组件例外。
- `void setSize(int width, int height) 1.1`

  使用给定的宽度和高度，重新设置组件的大小。
- `void setLocation(int x,int y) 1.1`

  将组件移到一个新的位置上。如果这个组件不是顶层组件，x和y坐标（或者p.x和p.y）是容器坐标；否则是屏幕坐标（例如：aJFrame）。
- `void setBounds(int x, int y, int width, int height) 1.1`

  移动并重新设置组件的大小。
- `Dimension getSize() 1.1`
- `void setSize(Dimension d) 1.1`

  获取或设置当前组件的size属性。

API java.awt.Window 1.0

- `void toFront()`

  将这个窗口显示在其他窗口前面。
- `void toBack()`

  将这个窗口移到桌面窗口栈的后面，并重新排列所有的可见窗口。
- `boolean isLocationByPlatform() 5.0`

- `void setLocationByPlatform(boolean b) 5.0`

  获取或设置locationByPlatform属性。这个属性在窗口显示之前被设置，由平台选择一个合适的位置。

API java.awt.Frame 1.0

- `boolean isResizable()`
- `void setResizable(boolean b)`

  获取或设置resizable属性。这个属性设置后，用户可以重新设置框架的大小。
- `String getTitle()`
- `void setTitle(String s)`

  获取或设置title属性，这个属性确定框架标题栏中的文字。
- `Image getIconImage()`
- `void setIconImage(Image image)`

  获取或设置iconImage属性，这个属性确定框架的图标。窗口系统可能会将图标作为框架装饰或其他部位的一部分显示。
- `boolean isUndecorated() 1.4`
- `void setUndecorated(boolean b) 1.4`

  获取或设置undecorated属性。这个属性设置后，框架显示中将没有标题栏或关闭按钮这样的装饰。在框架显示之前，必须调用这个方法。
- `int getExtendedState() 1.4`
- `void setExtendedState(int state) 1.4`

  获取或设置窗口状态。状态是下列值之一

```
Frame.NORMAL
Frame.ICONIFIED
Frame.MAXIMIZED_HORIZ
Frame.MAXIMIZED_VERT
Frame.MAXIMIZED_BOTH
```

API java.awt.Toolkit 1.0

- `static Toolkit getDefaultToolkit()`

  返回默认的工具箱。
- `Dimension getScreenSize()`

  返回用户屏幕的尺寸。
- `Image getImage(String filename)`

  加载文件名为filename的图像。

7.6 在组件中显示信息

本节将论述如何在框架内显示信息。例如，我们不再像第3章那样，采用文本方式将“Not a

Hello, World program”显示在控制台窗口中，而是在框架中显示这个消息，如图7-6所示。

可以将消息字符串直接绘制在框架中，但这并不是一种好的编程习惯。在Java中，框架被设计为放置组件的容器，可以将菜单栏和其他的用户界面元素放置在其中。在通常情况下，应该在另一组件上绘制信息，并将这个组件添加到框架中。

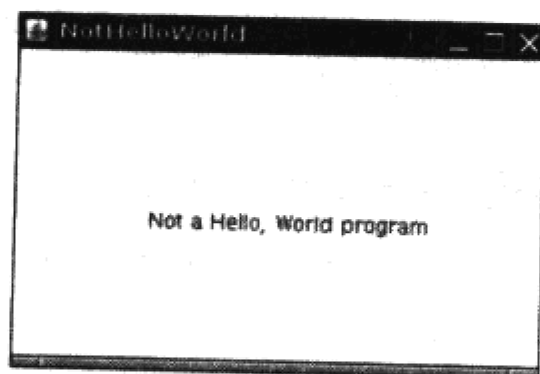


图7-6 一个显示消息的框架

JFrame的结构相当复杂。在图7-7中给出了JFrame的结构。可以看到，在JFrame中有四层面板。其中的根面板、层级面板和玻璃面板人们并不太关心；它们是用来组织菜单栏和内容窗格以及实现观感的。Swing程序员最关心的是*内容窗格*（content pane）。在设计框架的时候，要使用下列代码将所有的组件添加到内容窗格中：

```
Container contentPane = frame.getContentPane();
Component c = . . .;
contentPane.add(c);
```

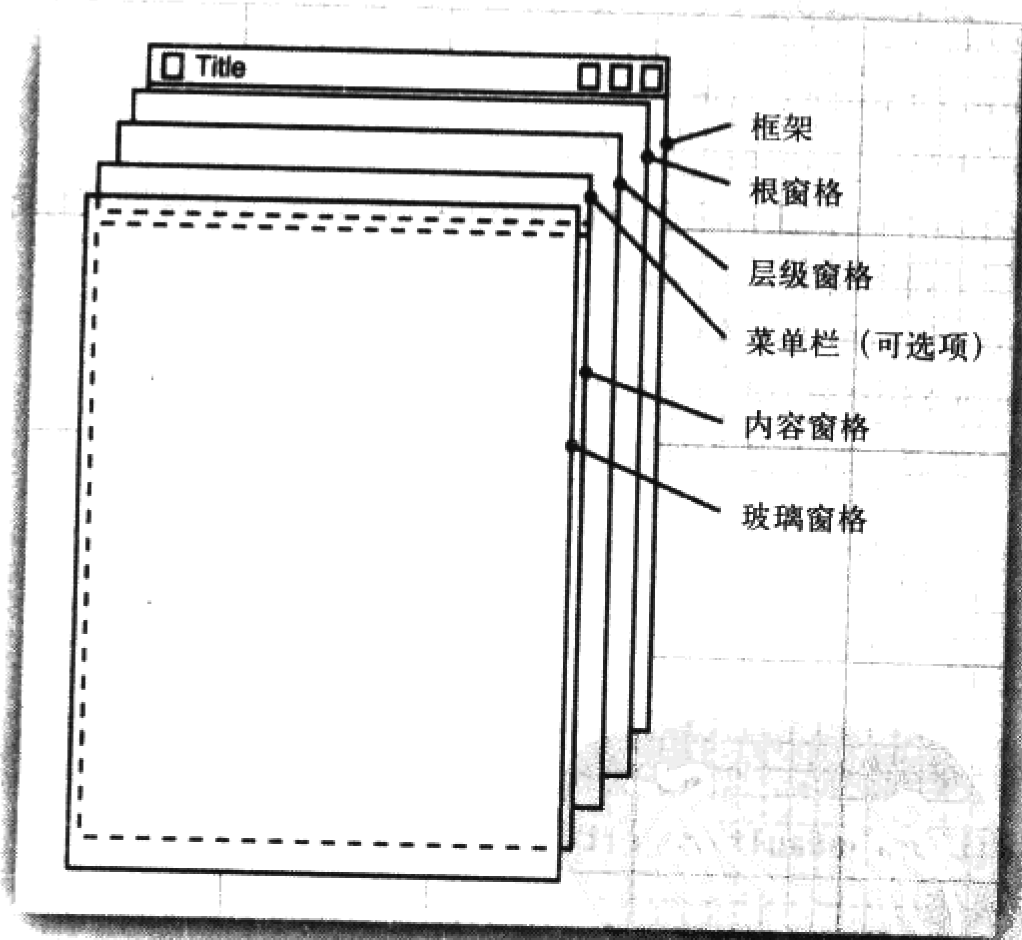


图7-7 JFrame的内部结构

在Java SE 1.4及以前的版本中，JFrame类中的add方法抛出了一个异常信息“Do not use JFrame.add().Use JFrame.getContentPane().add instead”。在Java SE 5.0中，JFrame.add方法不再显示这些提示信息，只是简单地调用内容窗格中的add。

因此，在Java SE 5.0中，可以直接调用

```
frame.add(c);
```

在这里，打算将一个绘制消息的组件添加到框架中。绘制一个组件，需要定义一个扩展JComponent的类，并覆盖其中的paintComponent方法。

paintComponent方法有一个Graphics类型的参数，这个参数保存着用于绘制图像和文本的设置，例如，设置的字体或当前的颜色。在Java中，所有的绘制都必须使用Graphics对象，其中包含了绘制图案、图像和文本的方法。

注释：Graphics参数与Windows中的设备环境或X11程序设计中的图形环境基本类似。

下列代码给出了如何创建一个能够进行绘制的组件：

```
class MyComponent extends JComponent
{
   public void paintComponent(Graphics g)
   {
      code for drawing
   }
}
```

无论何种原因，只要窗口需要重新绘图，事件处理器就会通告组件，从而引发执行所有组件的painComponent方法。

一定不要自己调用paintComponent方法。在应用程序需要重新绘图的时候，这个方法将被自动地调用，不要人为地干预这个自动的处理过程。

何种类别的动作会触发这个自动响应过程呢？例如，在用户扩大窗口或极小化窗口，然后又恢复窗口的大小时会引发重新绘图。如果用户弹出了另外一个窗口，并且这个窗口覆盖了一个已经存在的窗口，使得覆盖的窗口不可见，则此时被覆盖的应用程序窗口被破坏，需要重新绘制（图形系统不保存下面的像素）。当然，窗口第一次显示时，需要处理一些代码，主要包含确定绘制最初元素的方式以及位置。

提示：如果需要强制刷新屏幕，就需要调用repaint方法，而不是paintComponent方法。它将引发采用相应配置的Graphics对象调用所有组件的paintComponent方法。

从上述代码片段中可以看到，paintComponent方法只有一个Graphics类型的参数。对于屏幕显示来说，Graphics对象的度量单位是像素。坐标（0，0）指出所绘制组件表面的左上角。

显示文本是一种特殊的绘图。在Graphics类中有一个drawString方法，调用的语法格式为：

```
g.drawString(text, x, y)
```

在这里，打算在原始窗口大约水平1/4，垂直1/2的位置显示字符串“Not a Hello, World program”。现在，尽管不知道应该如何度量这个字符串的大小，但可以将字符串的开始位置定义在坐标（75，100）。这就意味着字符串中的第一个字符位于从左向右75个像素，从上向下100个像素的位置（实际上，文本的基线位于像素100的位置，有关文本的度量方式将在稍后阐述）。因此，paintComponent方法的书写内容如下所示：

```
class NotHelloWorldComponent extends JComponent
{
   public void paintComponent(Graphics g)
   {
      g.drawString("Not a Hello, World program", MESSAGE_X, MESSAGE_Y);
```

```
   }

   public static final int MESSAGE_X = 75;
   public static final int MESSAGE_Y = 100;
}
```



注释：有些程序员更喜欢扩展JPanel，而不是JComponent。JPanel是一个可以包含其他组件的容器（container），但同样也可以在其上面进行绘制。有一点不同之处是，面板不透明，这意味着需要在面板的边界内绘制所有的像素。最容易实现的方法是，在每个面板子类的paintComponent方法中调用super.paintComponent来用背景色绘制面板：

```
class NotHelloWorldPanel extends JPanel
{
   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);

      . . . // code for drawing will go here
   }
}
```

例7-3 NotHelloWorld.java

```
import javax.swing.*;
import java.awt.*;

/**
 * @version 1.32 2007-06-12
 * @author Cay Horstmann
 */
public class NotHelloWorld
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               NotHelloWorldFrame frame = new NotHelloWorldFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * A frame that contains a message panel
 */
class NotHelloWorldFrame extends JFrame
{
   public NotHelloWorldFrame()
   {
      setTitle("NotHelloWorld");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
```

```
      // add panel to frame

      NotHelloWorldPanel panel = new NotHelloWorldPanel();
      add(panel);
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;
}

/**
 * A panel that displays a message.
 */
class NotHelloWorldPanel extends JPanel
{
   public void paintComponent(Graphics g)
   {
      g.drawString("Not a Hello, World program", MESSAGE_X, MESSAGE_Y);
   }

   public static final int MESSAGE_X = 75;
   public static final int MESSAGE_Y = 100;
}

```

API javax.swing.JFrame 1.2

- Container getContentPane()

  返回这个JFrame的内容窗格对象。

- Component add(Component c)

  将一个给定的组件添加到该框架的内容窗格中（在Java SE 5.0以前的版本中，这个方法将抛出一个异常）。

API java.awt.Component 1.0

- void repaint()

  "尽可能快地"重新绘制组件。

- public void repaint(int x, int y, int width, int height)

  "尽可能快地"重新绘制组件的一部分。

API javax.swing.JComponent 1.2

- void paintComponent(Grphics g)

  覆盖这个方法来描述应该如何绘制自己的组件。

7.7 2D图形

自从Java版本1.0以来，Graphics类就包含绘制直线、矩形和椭圆等方法。但是，这些绘制

图形的操作能力非常有限。例如，不能改变线的粗细，不能旋转这些图形。

Java SE 1.2引入了Java 2D库，这个库实现了一组功能强大的图形操作。在本章中，只介绍Java 2D库的基础部分，有关高级功能的详细内容请参看卷II的高级AWT章节。

要想使用Java 2D库绘制图形，需要获得一个Graphics2D类对象。这个类是Graphics类的子类。自从Java SE 2版本以来，paintComponent方法就会自动地获得一个Graphics2D类对象，我们只需要进行一次类型转换就可以了。如下所示：

```
public void paintComponent(Graphics g)
{
   Graphics2D g2 = (Graphics2D) g;
   . . .
}
```

Java 2D库采用面向对象的方式将几何图形组织起来。包含描述直线、矩形的椭圆的类：

```
Line2D
Rectangle2D
Ellipse2D
```

这些类全部实现了Shape接口。

注释：Java 2D库支持更加复杂的图形，例如圆弧、二次曲线、三次曲线和通用路径。有关更详细的内容请参看卷II第7章。

要想绘制图形，首先要创建一个实现了Shape接口的类的对象，然后调用Graphics2D类中的draw方法。例如，

```
Rectangle2D rect = . . .;
g2.draw(rect);
```

注释：在Java 2D库出现之前，程序员使用Grpahics类中的drawRectangle方法绘制图形。从表面上看，老式风格的方法调用起来好像更加简单一点。然而，使用Java 2D库，可以选择Java 2D库中提供的一些工具提高绘制能力。

使用Java 2D图形类或许会增加一些复杂度。在1.0的绘制方法中，采用的是整型像素坐标，而Java 2D图形采用的是浮点坐标。在很多情况下，用户可以使用更有意义的形式（例如，微米或英寸）指定图形的坐标，然后再将其转换成像素，这样做很方便。在Java 2D库中，内部的很多浮点计算都采用单精度float。毕竟，几何计算的最终目的是要设置屏幕或打印机的像素，所以单精度完全可以满足要求了。只要舍入误差限制在一个像素的范围内，视觉效果就不会受到任何影响。另外，在某些平台上，float计算的速度比较快，并且只占据double值的一半存储量。

然而，有时候程序员处理float并不太方便，这是因为Java程序设计语言在将double值转换成float值时必须进行类型转换。例如，考虑下列的语句：

```
float f = 1.2; // Error
```

这条语句无法通过编译，因为常量1.2属于double类型，而编译器不允许丢失精度。解决的方法是给浮点常量添加一个后缀F：

```
float f = 1.2F; // Ok
```

现在，看一下这条语句：

```
Rectangle2D r = . . .
float f = r.getWidth(); // Error
```

这条语句也无法通过编译，其原因与前面一样。由于getWidth方法的返回类型是double，所以需要进行类型转换：

```
float f = (float) r.getWidth(); // Ok
```

由于后缀和类型转换都有点麻烦，所以2D库的设计者决定为每个图形类提供两个版本：一个是为那些节省空间的程序员提供的float类型的坐标；另一个是为那些懒惰的程序员提供的double类型的坐标（本书主要采用的是第二个版本，即double类型的坐标）。

这个库的设计者选择了一种古怪且在最初看起来还有些混乱的方式进行了打包。看一下Rectangle2D类，这是一个拥有两个具体子类的抽象类，这两个具体子类也是静态内部类：

```
Rectangle2D.Float
Rectangle2D.Double
```

图7-8显示了它们之间的继承示意图。

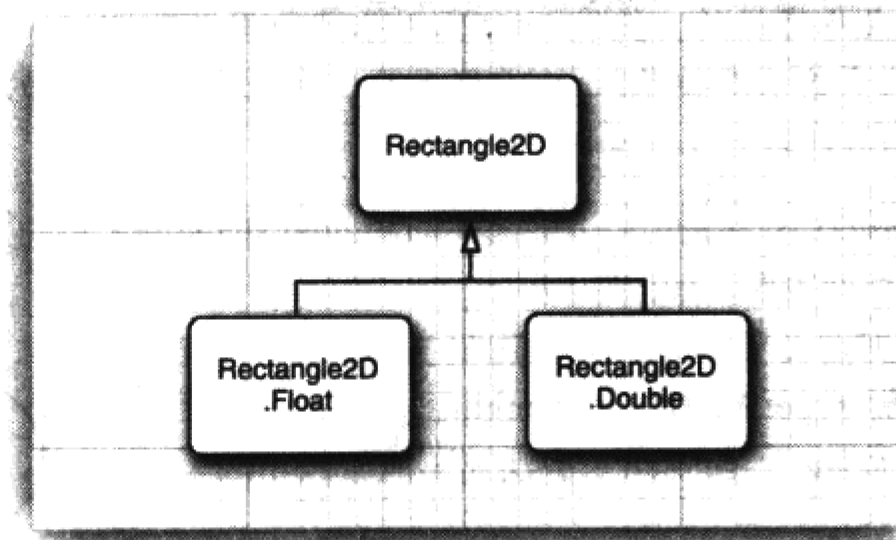


图7-8 2D矩形类

最好淡化这两个具体类是静态内部类，这样做只是为了避免使用FloatRectangle2D和DoubleRectangle2D这样的名字（有关静态内部类更详细的信息请参看第6章）。

当创建一个Rectangle2D.Float对象时，应该提供float型数值的坐标。而创建Rectangle2D.Double对象时，应该提供double型数值的坐标。

```
Rectangle2D.Float floatRect = new Rectangle2D.Float(10.0F, 25.0F, 22.5F, 20.0F);
Rectangle2D.Double doubleRect = new Rectangle2D.Double(10.0, 25.0, 22.5, 20.0);
```

实际上，由于Rectangle2D.Float和Rectangle2D.Double都扩展于Rectangle2D类，并且子类只覆盖了Rectangle2D超类中的方法，所以没有必要记住图形类型。可以直接使用Rectangle2D变量保存矩形的引用。

```
Rectangle2D floatRect = new Rectangle2D.Float(10.0F, 25.0F, 22.5F, 20.0F);
Rectangle2D doubleRect = new Rectangle2D.Double(10.0, 25.0, 22.5, 20.0);
```

也就是说，只有在构造图形对象时，才需要使用烦人的内部类。

构造参数表示矩形的左上角位置、宽和高。

✔ **注释**：实际上，Rectangle2D.Float类包含了一个不是由Rectangle2D继承而来的附加方法setRect(float x, float y, float h, float w)。如果将Rectangle2D.Float的引用存储在Rectangle2D变量中，那就会失去这个方法。但是，这也没有太大关系，因为在Rectangle2D中有一个参数为double类型的setRect方法。

Rectangle2D方法的参数和返回值均为double类型。例如，即使Rectangle2D.Float对象存储float类型的宽度，getWidth方法也返回一个double值。

! **提示**：直接使用Double图形类可以避免处理float类型的值，然而如果需要创建上千个图形对象，还是应该考虑使用Float类，这样可以节省存储空间。

前面对Rectangle2D类的论述也适用于其他图形类。另外，Point2D类也有两个子类Point2D.Float和Point2D.Double。下面是构造一个点对象的方法：

```
Point2D p = new Point2D.Double(10, 20);
```

! **提示**：Point2D类是很有用的。使用Point2D对象比使用单独的x和y更加具有面向对象的风格。许多构造器和方法都接收Point2D型参数，我们建议在可能的情况下使用Point2D对象。这样会使几何计算更容易理解。

Rectangle2D和Ellipse2D类都是由公共超类RectangularShape继承来的。无可非议，椭圆不是矩形，但它们都有着矩形边界，如图7-9所示。

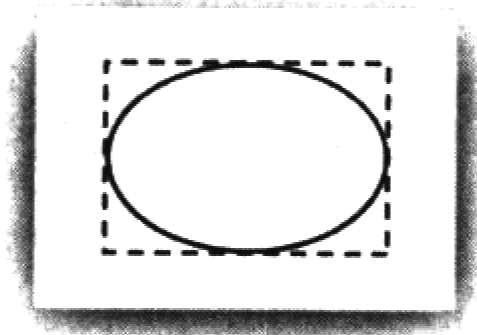


图7-9 椭圆的矩形边界

RectangularShape类定义了20多个有关图形操作的通用方法，其中比较常用的方法有getWidth、getHeight、getCenterX、getCenterY等（但在写本书时，getCenter方法还不能以Point2D对象的形式返回中心位置）。

最后，从Java 1.0遗留下来的两个类也被放置在图形类的继承层次中。它们是Rectangle和Point类，分别扩展于Rectangle2D和Point2D类，并用整型坐标存储矩形和点。

图7-10给出了图形类之间的关系。不过，省略了Double和Float子类。图中的遗留类采用填充灰色的方式标记。

Rectangle2D和Ellipse2D对象很容易构造。需要给出

- 左上角的x和y坐标；
- 宽和高。

对于椭圆，这些内容代表外接矩形。例如，

```
Ellipse2D e = new Ellipse2D.Double(150, 200, 100, 50);
```

用左上角位于（150，200）、宽100、高50的外接矩形构造一个椭圆。

然而，有时候并不知道左上角的位置。经常得到的是矩形的两个对角点，而这两个对角不一定是左上角和右下角。不能直接这样构造一个矩形：

```
Rectangle2D rect = new Rectangle2D.Double(px, py, qx - px, qy - py); // Error
```

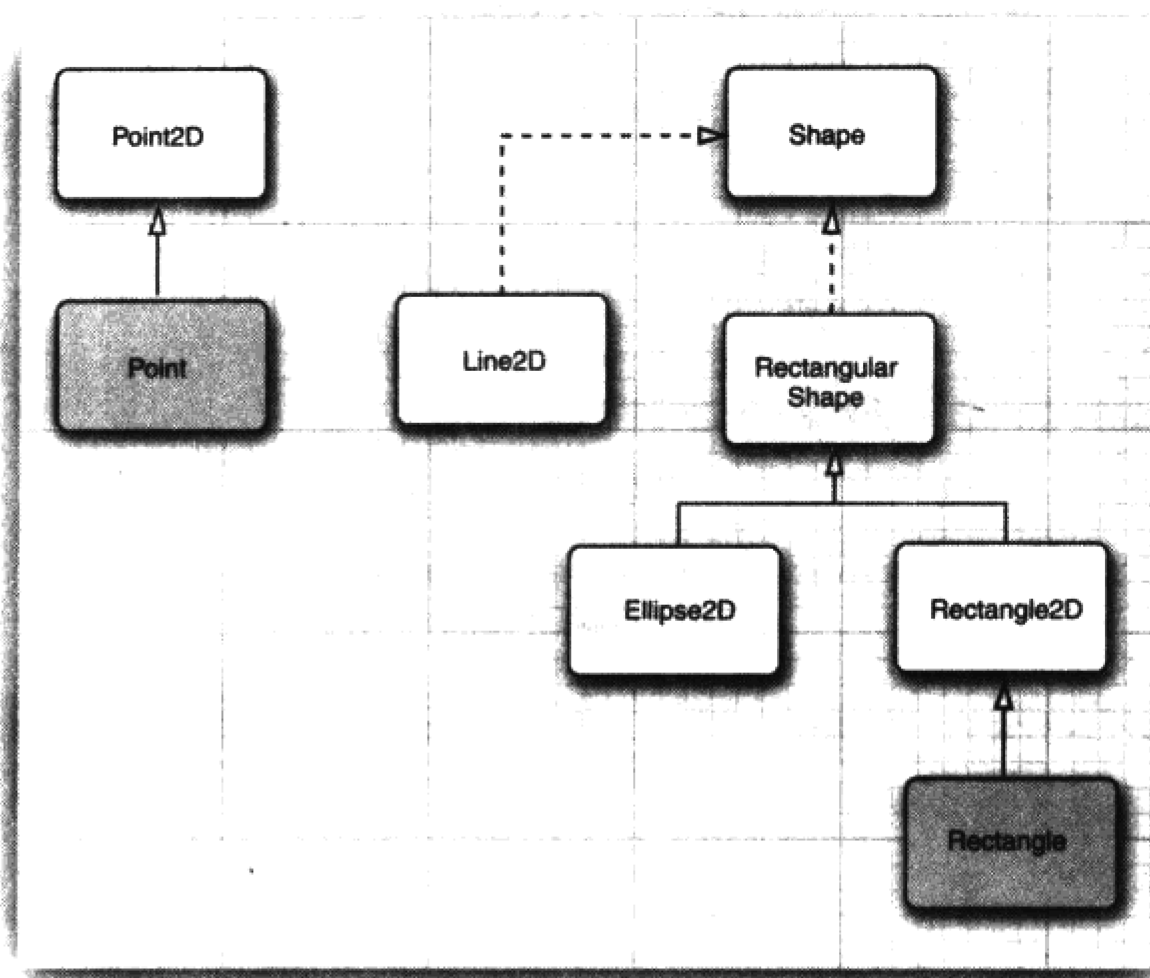


图7-10 图形类之间的关系

如果p不是左上角，两个坐标之差就为负，矩形就为空。在这种情况下，首先创建一个空矩形，然后调用setFrameFromDiagonal方法，如下所示：

```
Rectangle2D rect = new Rectangle2D.Double();
rect.setFrameFromDiagonal(px, py, qx, qy);
```

或者，如果已知的顶点分别用Point2D类型的两个对象p和q表示，就应该这样调用：

```
rect.setFrameFromDiagonal(p, q);
```

在构造椭圆（这种情况还出现在构造其他图形时）时，通常可以知道椭圆的中心、宽和高，而不是外接矩形的顶点。setFrameFromCenter方法使用中心点，但仍然要给出四个顶点中的一个。因此，通常采用下列方式构造椭圆：

```
Ellipse2D ellipse = new Ellipse2D.Double(centerX - width / 2, centerY - height / 2, width, height);
```

要想构造一条直线，需要提供起点和终点。这两个点既可以使用Point2D对象表示，也可以使用一对数值表示：

```
Line2D line = new Line2D.Double(start, end);
```

或者

```
Line2D line = new Line2D.Double(startX, startY, endX, endY);
```

例7-4中的程序绘制了一个矩形；这个矩形的内接椭圆；矩形的对角线以及以矩形中心为圆点的圆。图7-11显示了结果。

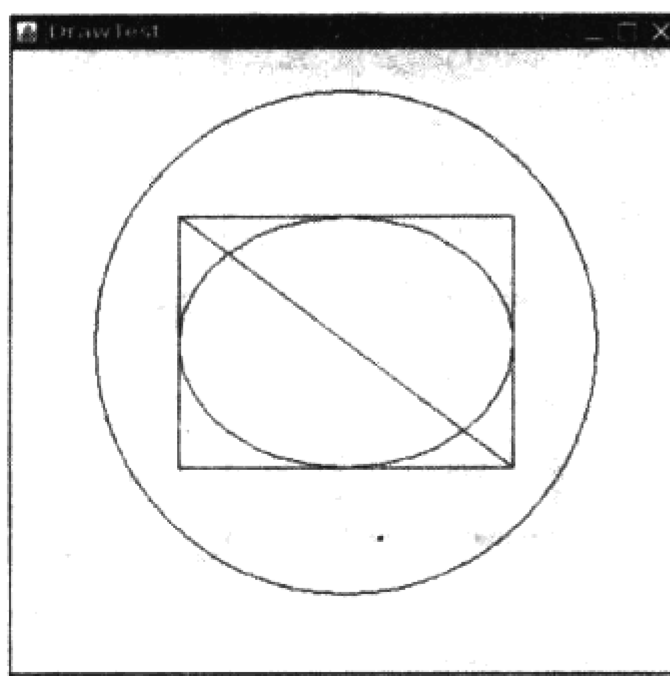


图7-11 绘制几何图形

例7-4 DrawTest.java

```
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

/**
 * @version 1.32 2007-04-14
 * @author Cay Horstmann
 */
public class DrawTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               DrawFrame frame = new DrawFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * A frame that contains a panel with drawings
 */
class DrawFrame extends JFrame
{
   public DrawFrame()
   {
      setTitle("DrawTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

```

```
        // add panel to frame

        DrawComponent component = new DrawComponent();
        add(component);
    }

    public static final int DEFAULT_WIDTH = 400;
    public static final int DEFAULT_HEIGHT = 400;
}

/**
 * A component that displays rectangles and ellipses.
 */
class DrawComponent extends JComponent
{
    public void paintComponent(Graphics g)
    {
        Graphics2D g2 = (Graphics2D) g;

        // draw a rectangle

        double leftX = 100;
        double topY = 100;
        double width = 200;
        double height = 150;

        Rectangle2D rect = new Rectangle2D.Double(leftX, topY, width, height);
        g2.draw(rect);

        // draw the enclosed ellipse

        Ellipse2D ellipse = new Ellipse2D.Double();
        ellipse.setFrame(rect);
        g2.draw(ellipse);

        // draw a diagonal line

        g2.draw(new Line2D.Double(leftX, topY, leftX + width, topY + height));

        // draw a circle with the same center

        double centerX = rect.getCenterX();
        double centerY = rect.getCenterY();
        double radius = 150;

        Ellipse2D circle = new Ellipse2D.Double();
        circle.setFrameFromCenter(centerX, centerY, centerX + radius, centerY + radius);
        g2.draw(circle);
    }
}
```


java.awt.geom.RectangularShape 1.2

- double getCenterX()
- double getCenterY()

- `double getMinX()`
- `double getMinY()`
- `double getMaxX()`
- `double getMaxY()`

返回闭合矩形的中心，以及最小、最大x和y坐标值。

- `double getWidth()`
- `double getHeight()`

返回闭合矩形的宽和高。

- `double getX()`
- `double getY()`

返回闭合矩形左上角的x和y坐标。

API java.awt.geom.Rectangle2D.Double 1.2

- `Rectangle2D.Double(double x,double y,double w,double h)`

利用给定的左上角、宽和高，构造一个矩形。

API java.awt.geom.Rectangle2D.Float 1.2

- `Rectangle2D.Float(float x,float y,float w,float h)`

利用给定的左上角、宽和高，构造一个矩形。

API java.awt.geom.Ellipse2D.Double 1.2

- `Ellipse2D.Double(double x,double y,double w,double h)`

利用给定的左上角、宽和高的外接矩形，构造一个椭圆。

API java.awt.geom.Point2D.Double 1.2

- `Point2D.Double(double x, double y)`

利用给定坐标构造一个点。

API java.awt.geom.Line2D.Double 1.2

- `Line2D.Double(Point2D start, Point2D end)`
- `Line2D.Double(double startX, double startY, double endX, double endY)`

使用给定的起点和终点，构造一条直线。

7.8 颜色

使用Graphics2D类的setPaint方法可以为图形环境上的所有后续的绘制操作选择颜色。例如：

```
g2.setPaint(Color.RED);
g2.drawString("Warning!", 100, 100);
```

只需要将调用draw替换为调用fill就可以用一种颜色填充一个封闭图形（例如：矩形或椭圆）的内部：

```
Rectangle2D rect = . . .;
g2.setPaint(Color.RED);
g2.fill(rect); // fills rect with red color
```

要想绘制多种颜色，就需要按照选择颜色、绘制图形、再选择另外一种颜色、再绘制图形的过程实施。

Color类用于定义颜色。在java.awt.Color类中提供了13个预定义的常量，它们分别表示13种标准颜色。

```
BLACK, BLUE, CYAN, DARK_GRAY, GRAY, GREEN, LIGHT_GRAY, MAGENTA, ORANGE, PINK, RED, WHITE, YELLOW
```

✔ **注释**：在Java SE 1.4之前的版本中，颜色常量的名字为小写形式，例如，Color.red。这似乎有些超出寻常，因为标准编码的惯例是采用大写形式书写常量。从Java SE 1.4开始，可以采用大写的形式书写标准颜色的名字，不过，为了向后兼容，也可以用小写形式书写。

可以通过提供红、绿和蓝三色成分来创建一个Color对象，以达到定制颜色的目的。三种颜色都是用0～255（也就是一个字节）之间的整型数值表示，调用Color的构造器格式为：

```
Color(int redness, int  greenness, int blueness)
```

下面是一个定制颜色的例子：

```
g2.setPaint(new Color(0, 128, 128)); // a dull blue-green
g2.drawString("Welcome!", 75, 125);
```

✔ **注释**：除了纯色以外，还可以选择更复杂的“颜料”设置，例如，改变色调（hue）或者图像。有关这方面更加详细的内容请参看卷II中的高级AWT章节。如果使用Graphics对象，而不是Graphics2D对象，就需要使用setColor方法设置颜色。

要想设置背景颜色，就需要使用Component类中的setBackground方法。Component类是JComponent类的祖先。

```
MyComponent p = new MyComponent();
p.setBackground(Color.PINK);
```

另外，还有一个setForeground方法，它是用来设定在组件上进行绘制时使用的默认颜色。

! **提示**：从名字就可以看出，Color类中的brighter( )方法和darker( )方法的功能，它们分别加亮或变暗当前的颜色。使用brighter方法也是加亮条目的好办法。实际上，brighter( )只微微地加亮一点。要达到耀眼的效果，需要调用三次这个方法：c.brighter( ). brighter( ). brighter( )。

Java在SystemColor类中预定义了很多颜色的名字。在这个类中的常量，封装了用户系统的各个元素的颜色。例如，

```
p.setBackground(SystemColor.window)
```

它将把面板的背景颜色设定为用户桌面上所有窗口使用的默认颜色。（无论何时重新绘制窗口，

都会填充背景颜色。）当希望让绘制的用户界面元素与用户桌面上已经存在的其他元素的颜色匹配时，使用SystemColor类中的颜色非常有用。表7-1列出了系统颜色的名字和它们的含义。

表7-1 系统颜色

desktop	桌面的背景颜色	textText	文本的前景颜色
activeCaption	标题的背景颜色	textInactiveText	非活动组件的文本颜色
activeCaptionText	标题的文本颜色	textHighlight	高亮度文本的背景颜色
activeCaptionBorder	标题文本的边框颜色	textHighlightText	高亮度文本的文本颜色
inactiveCaption	非活动标题的背景颜色	control	组件的背景颜色
inactiveCaptionText	非活动标题的文本颜色	controlText	组件的文本颜色
inactiveCaptionBorder	非活动标题的边框颜色	controlLtHighlight	组件的浅高亮度颜色
Window	窗口的背景	controlHighlight	组件的高亮度颜色
windowBorder	窗口边框的颜色	controlShadow	组件的阴影颜色
windowText	窗口内的文本颜色	controlDkShadow	组件的暗阴影颜色
menu	菜单的背景颜色	scrollbar	滚动条的背景颜色
menuText	菜单的文本颜色	info	帮助区文本的颜色
text	文本的背景颜色	infoText	帮助区的文本颜色

API java.awt.Color 1.0

- `Color(int r,int g,int b)`

  创建一个颜色对象。

  参数：r　红色值（0-255）

  　　　g　绿色值（0-255）

  　　　b　蓝色值（0-255）

- `Color getColor()`

API java.awt.Graphics 1.0

- `void setColor(Color c)`

  获取或改变当前的颜色。所有后续的绘图操作都使用这个新颜色。

  参数：c　新颜色

- `Paint getPaint()`

API java.awt.Graphics2D 1.2

- `void setPaint(Paint p)`

  获取或设置这个图形环境的绘制属性。Color类实现了Paint接口。因此，可以使用这个方法将绘制属性设置为纯色。

- `void fill(Shape s)`

  用当前的颜料填充该图形。

API java.awt.geom.Line2D.Double 1.2


java.awt.Component 1.0

- Color getBackground()
- void setBackground(Color c)

  获取或设置背景颜色。

  参数：c　　新背景颜色
- Color getForeground()
- void setForeground(Color c)

  获取或设置前景颜色。

  参数：c　　新前景颜色

7.9 为文本设定特殊字体

在本章开始的"Not a Hello, World"程序中用默认字体显示了一个字符串。实际上，经常希望选用不同的字体显示文本。人们可以通过字体名（font face name）指定一种字体。字体名由"Helvetica"这样的字体家族名（font family name）和一个可选的"Bold"后缀组成。例如，"Helvetica"和"Helvetica Bold"属于"Helvetica"家族的字体。

要想知道某台特定计算机上允许使用的字体，就需要调用GraphicsEnvironment类中的getAvailableFontFamilyNames方法。这个方法将返回一个字符型数组，其中包含了所有可用的字体名。GraphicsEnvironment类描述了用户系统的图形环境，为了得到这个类的对象，需要调用静态的getLocalGraphicsEnvironment方法。下面这个程序将打印出系统上的所有字体名：

```
import java.awt.*;

public class ListFonts
{
   public static void main(String[] args)
   {
      String[] fontNames = GraphicsEnvironment
         .getLocalGraphicsEnvironment()
         .getAvailableFontFamilyNames();
      for (String fontName : fontNames)
         System.out.println(fontName);
   }
}
```

在某个系统上，输出的结果为：

```
Abadi MT Condensed Light
Arial
Arial Black
Arial Narrow
Arioso
Baskerville
Binner Gothic
. . .
```

后面还有70种左右的字体。

字体名可以商标化，字体设计在一些权限内可以版权化。因此，字体的分发需要向字体的

创始者付版税。当然，像名牌香水有廉价仿制品一样，字体也有外观相似的。例如，Helvetica的仿制品就是Windows中称为Arial的字体。

为了创建一个公共基准，AWT定义了五个逻辑（logical）字体名：

```
SansSerif
Serif
Monospaced
Dialog
DialogInput
```

这些字体将被映射到客户机上的实际字体。例如，在Windows系统中，SansSerif将被映射到Arial上。

另外，Sun JDK包含3种字体，它们是“Lucida Sans”，“Lucida Bright”和 “Lucida Sans Typewriter”。

要想使用某种字体绘制字符，必须首先利用指定的字体名、字体风格和字体大小来创建一个Font类对象。下面是构造一个Font对象的例子：

```
Font sansbold14 = new Font("SansSerif", Font.BOLD, 14);
```

第三个参数是以点数目计算的字体大小。点数目是排版中普遍使用的表示字体大小的单位，每英寸包含72个点。

在Font构造器中，提供字体名的位置也可以给出逻辑字体名称。另外，利用Font构造器的第二个参数可以指定字体的风格（常规、加粗、斜体或加粗斜体），下面是几个字体风格的值：

```
Font.PLAIN
Font.BOLD
Font.ITALIC
Font.BOLD + Font.ITALIC
```

注释：字体映射定义在Java安装的jre/lib子目录中的fontconfig.properties文件中。有关这个文件的详细内容请参看http://java.sun.com/javase/6/docs/guide/intl/fontconfig.html。

可以读取TrueType或PostScriot Type 1格式的字体文件。这需要一个字体输入流——通常从磁盘文件或者URL读取（有关流的更详细信息请参看卷II第1章）。然后调用静态方法Font.createFont：

```
URL url = new URL("http://www.fonts.com/Wingbats.ttf");
InputStream in = url.openStream();
Font f1 = Font.createFont(Font.TRUETYPE_FONT, in);
```

上面定义的字体为常规字体，大小为1。可以使用deriveFont方法得到希望大小的字体：

```
Font f = f1.deriveFont(14.0F);
```

警告：deriveFont方法有两个重载版本。一个（有一个float参数）设置字体的大小；另一个（有一个int参数）设置字体风格。所以f.deriveFont(14)设置的是字体风格，而不是大小（其结果为斜体，因为14的二进制表示的是ITALIC，而不是BOLD）。

Java字体包含了通用的ASCII字符和符号。例如，如果用Dialog字体打印字符‘\u2297’，那么就会看到⊗字符。只有在Unicode字符集中定义的符号才能够使用。

下面这段代码将使用系统中14号加粗的标准sans serif字体显示字符串“Hello,World”：

```
Font sansbold14 = new Font("SansSerif", Font.BOLD, 14);
g2.setFont(sansbold14);
String message = "Hello, World!";
g2.drawString(message, 75, 100);
```

接下来，将字符串绘制在面板的中央，而不是任意位置。因此，需要知道字符串占据的宽和高的像素数量。这两个值取决于下面三个因素：

- 使用的字体（在前面列举的例子中为sans serif，加粗，14号）；
- 字符串（在前面列举的例子中为”Hello,World”）；
- 绘制字体的设备（在前面列举的例子中为用户屏幕）。

要想得到屏幕设备字体属性的描述对象，需要调用Graphics2D类中的getFontRenderContext方法。它将返回一个FontRenderContext类对象。可以直接将这个对象传递给Font类的getStringBounds方法：

```
FontRenderContext context = g2.getFontRenderContext();
Rectangle2D bounds = f.getStringBounds(message, context);
```

getStringBounds方法将返回包围字符串的矩形。

为了解释这个矩形的大小，需要清楚几个排版的相关术语。如图7-12所示。*基线*（baseline）是一条虚构的线，例如，字母“e”所在的底线。*上坡度*（ascent）是从基线到*坡顶*（ascenter）的距离。例如，“b”和“k”以及大写字母的上面部分。*下坡度*（descent）是从基线到*坡底*（descenter）的距离，坡底是“p”和“g”这种字母的底线。

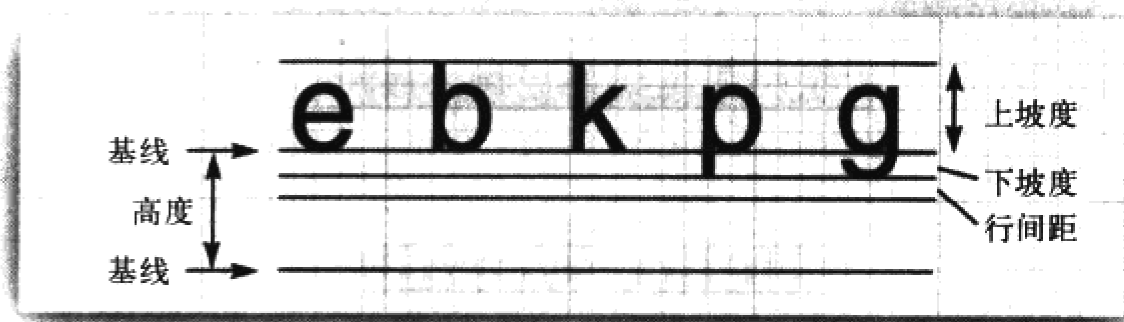


图7-12 排版术语解释

行间距（leading）是某一行的坡底与其下一行的坡顶之间的空隙（这个术语源于排字机分隔行的引导带）。字体的高度是连续两个基线之间的距离，它等于下坡度+行间距+上坡度。

getStringBounds方法返回的矩形宽度是字符串水平方向的宽度。矩形的高度是上坡度、下坡度、行间距的总和。这个矩形始于字符串的基线，矩形顶部的y坐标为负值。因此，可以采用下面的方法获得字符串的宽度、高度和上坡度：

```
double stringWidth = bounds.getWidth();
double stringHeight = bounds.getHeight();
double ascent = -bounds.getY();
```

如果需要知道下坡度或行间距，可以使用Font类的getLineMetrics方法。这个方法将返回一个LineMetrics类对象，获得下坡度和行间距的方法是：

```
LineMetrics metrics = f.getLineMetrics(message, context);
```

```
float descent = metrics.getDescent();
float leading = metrics.getLeading();
```

下面这段代码使用了所有这些信息，将字符串显示在包围它的组件中央：

```
FontRenderContext context = g2.getFontRenderContext();
Rectangle2D bounds = f.getStringBounds(message, context);

// (x,y) = top left corner of text
double x = (getWidth() - bounds.getWidth()) / 2;
double y = (getHeight() - bounds.getHeight()) / 2;

// add ascent to y to reach the baseline
double ascent = -bounds.getY();
double baseY = y + ascent;
g2.drawString(message, (int) x, (int) baseY);
```

为了能够获得中央的位置，可以使用getWidth( )得到组件的宽度。使用bounds.getWidth( )得到字符串的宽度。前者减去后者就是两侧应该剩余的空间。因此，每侧剩余的空间应该是这个差值的一半。高度也是一样。

注释：如果需要在paintComponent方法外部计算布局图的尺度，不能从Graphics2D对象得到字体绘制环境。换作调用JComponent类的getFontMetrics方法，而后紧接着调用getFontRenderContext:

```
FontRenderContext context = getFontMetrics(f).getFontRenderContext();
```

为了说明位置是正确的，示例程序绘制了基线和包围这个字符串的矩形。图7-13给出了屏幕显示结果。例7-5是程序清单。

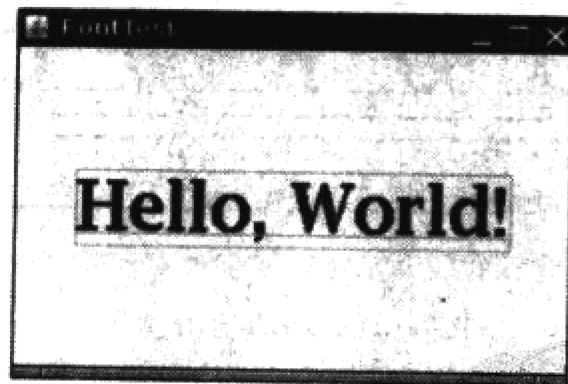


图7-13 绘制基线和字符串边框

例7-5 FontTest.java

```
import java.awt.*;
import java.awt.font.*;
import java.awt.geom.*;
import javax.swing.*;

/**
 * @version 1.33 2007-04-14
 * @author Cay Horstmann
 */
public class FontTest
{
```

```
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               FontFrame frame = new FontFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * A frame with a text message component
 */
class FontFrame extends JFrame
{
   public FontFrame()
   {
      setTitle("FontTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // add component to frame

      FontComponent component = new FontComponent();
      add(component);
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;
}

/**
 * A component that shows a centered message in a box.
 */
class FontComponent extends JComponent
{
   public void paintComponent(Graphics g)
   {
      Graphics2D g2 = (Graphics2D) g;

      String message = "Hello, World!";

      Font f = new Font("Serif", Font.BOLD, 36);
      g2.setFont(f);

      // measure the size of the message

      FontRenderContext context = g2.getFontRenderContext();
      Rectangle2D bounds = f.getStringBounds(message, context);

      // set (x,y) = top-left corner of text

      double x = (getWidth() - bounds.getWidth()) / 2;
      double y = (getHeight() - bounds.getHeight()) / 2;
```

```

        // add ascent to y to reach the baseline

        double ascent = -bounds.getY();
        double baseY = y + ascent;

        // draw the message

        g2.drawString(message, (int) x, (int) baseY);

        g2.setPaint(Color.LIGHT_GRAY);

        // draw the baseline

        g2.draw(new Line2D.Double(x, baseY, x + bounds.getWidth(), baseY));

        // draw the enclosing rectangle

        Rectangle2D rect = new Rectangle2D.Double(x, y, bounds.getWidth(), bounds.getHeight());
        g2.draw(rect);
    }
}
```

API `java.awt.Font 1.0`

- `Font(String name, int style, int size)`

  创建一个新字体对象。

  参数：name　字体名。不是字体名（例如，“Helvetica Bold”），就是逻辑字体名（例如，“Serif”、“SansSerif”）

  style　字体风格（Font.PLAIN、Font.BOLD、Font.ITALIC或Font.BOLD+Font.ITALIC）

  size　字体大小（例如，12）

- `String getFontName()`

  返回字体名，例如，“Helvetica Bold”。

- `String getFamily()`

  返回字体家族名，例如，“Helvetica”。

- `String getName()`

  如果采用逻辑字体名创建字体，将返回逻辑字体，例如，“SansSerif”，否则，返回字体名。

- `Rectangle2D getStringBounds(String s, FontRenderContext context) 1.2`

  返回包围这个字符串的矩形。矩形的起点为基线。矩形顶端的y坐标等于上坡度的负值。矩形的高度等于上坡度、下坡度和行间距之和。宽度等于字符串的宽度。

- `LineMetrics getLineMetrics(String s, FontRenderContext context) 1.2`

  返回测定字符串宽度的一个线性metrics对象。

- `Font deriveFont(int style) 1.2`
- `Font deriveFont(float size) 1.2`

- `Font deriveFont(int style, float size)` 1.2

  返回一个新字体，除给定大小和字体风格外，其余与原字体一样。

API java.awt.font.LineMetrics 1.2

- `float getAscent()`

  返回字体的上坡度——从基线到大写字母顶端的距离。
- `float getDescent()`

  返回字体的下坡度——从基线到坡底的距离。
- `float getLeading()`

  返回字体的行间距——从一行文本底端到下一行文本顶端之间的空隙。
- `float getHeight()`

  返回字体的总高度——两条文本基线之间的距离（下坡度+行间距+上坡度）。

API java.awt.Graphics 1.0

- `Font getFont()`
- `void setFont(Font font)`

  获取或设置当前的字体。这种字体将被应用于后续的文本绘制操作中。

  参数：font　　一种字体
- `void drawString(String str, int x, int y)`

  采用当前字体和颜色绘制一个字符串。

  参数：str　　将要绘制的字符串

  　　　x　　字符串开始的x坐标

  　　　y　　字符串基线的y坐标

API java.awt.Graphics2D 1.2

- `FontRenderContext getFontRenderContext()`

  返回这个图形文本中，指定字体特征的字体绘制环境。
- `void drawString(String str, float x, float y)`

  采用当前的字体和颜色绘制一个字符串。

  参数：str　　将要绘制的字符串

  　　　x　　字符串开始的x坐标

  　　　y　　字符串基线的y坐标

API javax.swing.JComponent 1.2

- `FontMetrics getFontMetrics(Font f)` 5.0

  获取给定字体的度量。FontMetrics类是LineMetrics类的早先版。

API java.awt.FontMetrics 1.0

- `FontRenderContext getFontRenderContext() 1.2`

  返回字体的字体绘制环境。

7.10 图像

到目前为止，我们已经看到了如何通过绘制直线和图形创建一个简单的图像。而对于照片这样的复杂图像来说，通常都是由扫描仪或特殊的图像处理软件生成的（正像在卷II中将看到的，逐像素地生成图像，并将结果存储到数组中也是可以的。这种方式通常用于生成不规则碎片的图像）。

一旦图像保存在本地文件或因特网的某个位置上，就可以将它们读到Java应用程序中，并在Graphics对象上进行显示。在Java SE 1.4中，读取一个图像十分简单。如果图像存储在本地文件中，就应该调用：

```
String filename = "...";
Image image = ImageIO.read(new File(filename));
```

否则，应该提供URL：

```
String urlname = "...";
Image image = ImageIO.read(new URL(urlname));
```

如果图像不可用，read方法将抛出一个IOException。在第11章中，将讨论有关异常处理的问题。而在目前的例子程序中只捕获异常，并打印出栈的轨迹。

这里的变量image包含了一个封装图像数据的对象引用。可以使用Graphics类的drawImage方法将图像显示出来。

```
public void paintComponent(Graphics g)
{
   . . .
   g.drawImage(image, x, y, null);
}
```

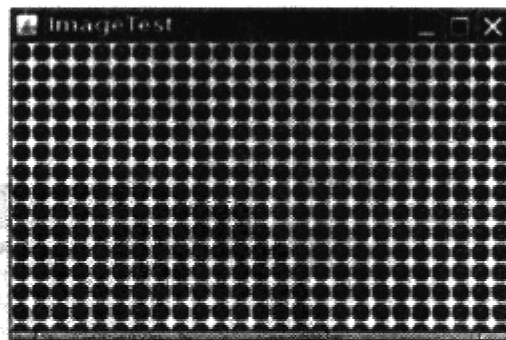


图7-14 平铺图像的窗口

例7-6又前进了一步，它在一个窗口中平铺显示了一幅图像。屏幕显示的结果如图7-14所示。这里采用paintComponent方法来实现平铺显示。它的基本过程为：先在左上角显示图像的一个拷贝，然后使用copyArea将其拷贝到整个窗口：

```
for (int i = 0; i * imageWidth <= getWidth(); i++)
  for (int j = 0; j * imageHeight <= getHeight(); j++)
     if (i + j > 0)
        g.copyArea(0, 0, imageWidth, imageHeight, i * imageWidth, j * imageHeight);
```

例7-6列出了图像显示程序的完整代码。

例7-6 ImageTest.java

```
import java.awt.*;
import java.io.*;
import javax.imageio.*;
```

```
import javax.swing.*;

/**
 * @version 1.33 2007-04-14
 * @author Cay Horstmann
 */
public class ImageTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               ImageFrame frame = new ImageFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * A frame with an image component
 */
class ImageFrame extends JFrame
{
   public ImageFrame()
   {
      setTitle("ImageTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // add component to frame

      ImageComponent component = new ImageComponent();
      add(component);
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;
}

/**
 * A component that displays a tiled image
 */
class ImageComponent extends JComponent
{
   public ImageComponent()
   {
      // acquire the image
      try
      {
         image = ImageIO.read(new File("blue-ball.gif"));
      }
      catch (IOException e)
      {
         e.printStackTrace();
```

```
      }
   }

   public void paintComponent(Graphics g)
   {
      if (image == null) return;

      int imageWidth = image.getWidth(this);
      int imageHeight = image.getHeight(this);

      // draw the image in the top-left corner

      g.drawImage(image, 0, 0, null);
      // tile the image across the component

      for (int i = 0; i * imageWidth <= getWidth(); i++)
         for (int j = 0; j * imageHeight <= getHeight(); j++)
            if (i + j > 0) g.copyArea(0, 0, imageWidth, imageHeight, i * imageWidth, j
                  * imageHeight);
   }

   private Image image;
}
```

API javax.imageio.ImageIO 1.4

- static BufferedImage read(File f)
- static BufferedImage read(URL u)

  从给定文件或URL上读取图像。

API java.awt.Graphics 1.0

- boolean drawImage(Image img, int x, int y, ImageObserver observer)

  绘制一幅非比例图像。注意：这个调用可能会在图像还没有绘制完毕就返回。

  参数：img 将要绘制的图像
  x 左上角的x坐标
  y 左上角的y坐标
  observer 绘制进程中以通告为目的的对象（可能为null）。

- boolean drawImage(Image img,int x,int y,int width,int height,ImageObserver observer)

  绘制一幅比例图像。系统按照比例将图像放入给定宽和高的区域。注意：这个调用可能会在图像还没有绘制完毕就返回。

  参数：img 将要绘制的图像
  x 左上角的x坐标
  y 左上角的y坐标
  width 描述图像的宽度

height　　描述图像的高度

observer　绘制进程中以通告为目的的对象（可能为null）

• `void copyArea(int x,int y,int width,int height,int dx,int dy)`

拷贝屏幕的一块区域。

参数：x　　原始区域左上角的x坐标

y　　原始区域左上角的y坐标

width　　原始区域的宽度

height　　原始区域的高度

dx　　原始区域到目标区域的水平距离

dy　　原始区域到目标区域的垂直距离

本章总结了有关Java图形编程的内容介绍。关于更高级的技术，可以参看卷II中有关2D图形和图像的操作。在下一章，读者将学习如何让程序对用户的输入进行响应。

第8章 事件处理

▲ 事件处理基础
▲ 动作
▲ 鼠标事件
▲ AWT事件继承层次

对于图形用户界面的程序来说，事件处理是十分重要的。要想实现用户界面，必须掌握Java事件处理的基本方法。本章将讲解Java AWT事件模型的工作机制，从中可以看到如何捕获用户界面组件和输入设备产生的事件。另外，本章还介绍如何以更加结构化的方式处理*动作*（actions）事件。

8.1 事件处理基础

任何支持GUI的操作环境都要不断地监视敲击键盘或点击鼠标这样的事件。操作环境将这些事件报告给正在运行的应用程序。如果有事件产生，每个应用程序将决定如何对它们做出响应。在Visual Basic这样的语言中，事件与代码之间有着明确的对应关系。程序员对相关的特定事件编写代码，并将这些代码放置在过程中，通常人们将它们称为*事件过程*（event procedure）。例如，有一个名为HelpButton的Visual Basic按钮有一个与之关联的HelpButton_Click事件过程。这个过程中的代码将在点击按钮后执行。每个Visual Basic的GUI组件都响应一个固定的事件集，不可能改变Visual Basic组件响应的事件集。

另一方面，如果使用像原始的C这样的语言进行事件驱动的程序设计，那就需要编写代码来不断地检查事件队列，以便查询操作环境报告的内容（通常这些代码被放置在包含很多switch语句的循环体中）。显然，这种方式编写的程序可读性很差，而且在有些情况下，编码的难度也非常大。它的好处在于响应的事件不受限制，而不像Visual Basic这样的语言，将事件队列对程序员隐藏起来。

Java程序设计环境折中了Visual Basic与原始C的事件处理方式，因此，它既有着强大的功能，又具有一定的复杂性。在AWT所知的事件范围内，完全可以控制事件从*事件源*（event source）例如，按钮或滚动条，到*事件监听器*（event listener）的传递过程，并将任何对象指派给事件监听器。不过事实上，应该选择一个能够便于响应事件的对象。这种*事件委托模型*（event delegation model）与Visual Basic那种预定义监听器模型比较起来更加灵活。

事件源有一些向其注册事件监听器的方法。当某个事件源产生事件时，事件源会向为事件注册的所有事件监听器对象发送一个通告。

像Java这样的面向对象语言，都将事件的相关信息封装在一个*事件对象*（event object）中。在Java中，所有的事件对象都最终派生于java.util.EventObject类。当然，每个事件类型还有子类，例如，ActionEvent和WindowEvent。

不同的事件源可以产生不同类别的事件。例如，按钮可以发送一个ActionEvent对象，而窗口可以发送WindowEvent对象。

综上所述，下面给出AWT事件处理机制的概要：

- 监听器对象是一个实现了特定监听器接口（listener interface）的类的实例。
- 事件源是一个能够注册监听器对象并发送事件对象的对象。
- 当事件发生时，事件源将事件对象传递给所有注册的监听器。
- 监听器对象将利用事件对象中的信息决定如何对事件做出响应。

图8-1显示了事件处理类和接口之间的关系。

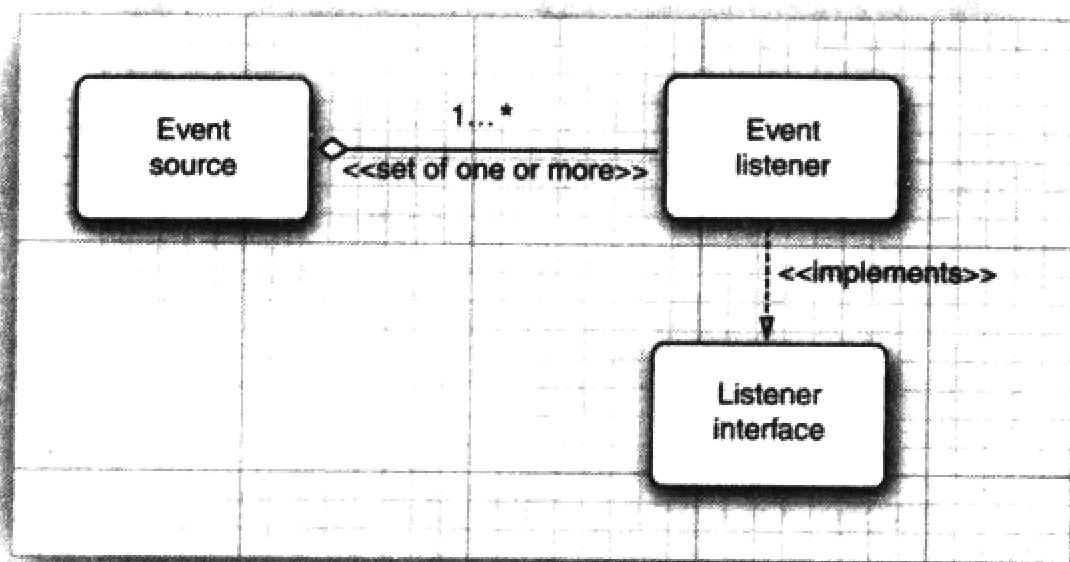


图8-1 事件源和监听器之间的关系

下面是监听器的一个示例：

```
ActionListener listener = . . .;
JButton button = new JButton("Ok");
button.addActionListener(listener);
```

现在，只要按钮产生了一个"动作事件"，listener对象就会得到通告。对于按钮来说，正像我们所想到的，动作事件就是点击按钮。

为了实现ActionListener接口，监听器类必须有一个被称为actionPerformed的方法，该方法接收一个ActionEvent对象参数。

```
class MyListener implements ActionListener
{
   . . .
   public void actionPerformed(ActionEvent event)
   {
      // reaction to button click goes here
      . . .
   }
}
```

只要用户点击按钮，JButton对象就会创建一个ActionEvent对象，然后调用listener.actionPerformed (event) 传递事件对象。可以将多个监听器对象添加到一个像按钮这样的事件源中。这样一来，只要用户点击按钮，按钮就会调用所有监听器的actionPerformed方法。

图8-2显示了事件源、事件监听器和事件对象之间的协作关系。

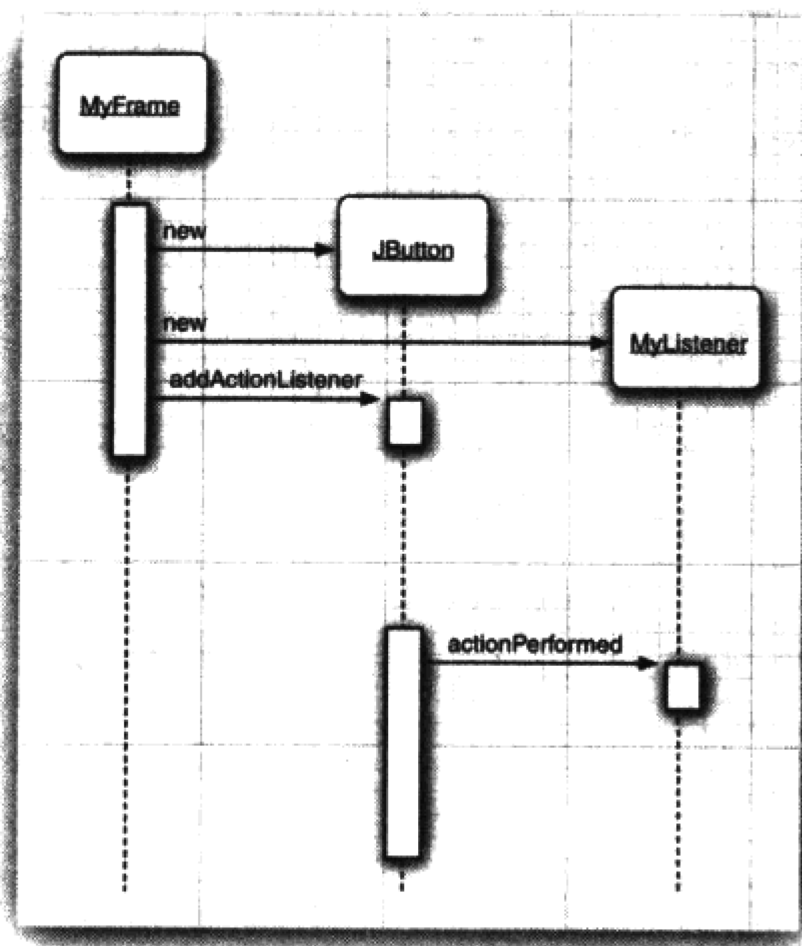


图8-2 事件通告

8.1.1 实例：处理按钮点击事件

为了加深对事件委托模型的理解，下面以一个响应按钮点击事件的简单示例来说明所需要知道的所有细节。在这个示例中，想要在一个面板中放置三个按钮，添加三个监听器对象用来作为按钮的动作监听器。

在这个情况下，只要用户点击面板上的任何一个按钮，相关的监听器对象就会接收到一个Action Event对象，它表示有个按钮被点击了。在示例程序中，监听器对象将改变面板的背景颜色。

在演示如何监听按钮点击事件之前，首先需要讲解一下如何创建按钮以及如何将它们添加到面板中（有关GUI元素更加详细的内容请参看第9章）。

可以通过在按钮构造器中指定一个标签字符串、一个图标或两项都指定来创建一个按钮。下面是两个示例：

```
JButton yellowButton = new JButton("Yellow");
JButton blueButton = new JButton(new ImageIcon("blue-ball.gif"));
```

将按钮添加到面板中需要调用add方法：

```
JButton yellowButton = new JButton("Yellow");
JButton blueButton = new JButton("Blue");
JButton redButton = new JButton("Red");

buttonPanel.add(yellowButton);
buttonPanel.add(blueButton);
buttonPanel.add(redButton);
```

图8-3显示了结果。

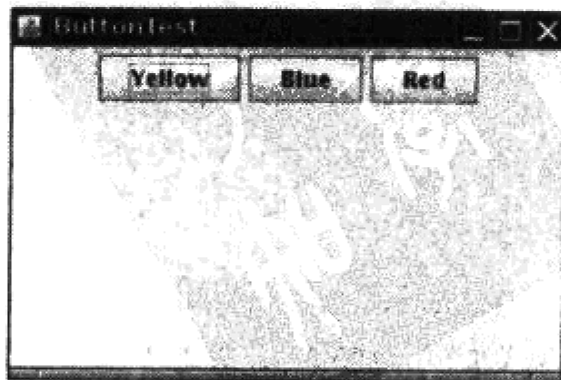


图8-3 填充按钮的面板

至此，知道了如何将按钮添加到面板上，接下来需要增加让面板监听这些按钮的代码。这需要一个实现了ActionListener接口的类。如前所述，应该包含一个actionPerformed方法，其签名为：

```
public void actionPerformed(ActionEvent event)
```

 **注释**：在按钮示例中，使用的ActionListener接口并不仅限于按钮点击事件。它可以应用于很多情况：

- 当采用鼠标双击的方式选择了列表框中的一个选项时；
- 当选择一个菜单项时；
- 当在文本域中敲击ENTER键时；
- 对于一个Timer组件来说，当到达指定的时间间隔时。

在本章和下一章中，读者将会看到更加详细的内容。

在各种情况下，使用ActionListener接口的方式都是一样的：actionPerformed方法（ActionListener中的唯一方法）将接收一个ActionEvent类型的对象作为参数。这个事件对象包含了事件发生时的相关信息。

当按钮被点击时，希望将面板的背景颜色设置为指定的颜色。这个颜色存储在监听器类中：

```
class ColorAction implements ActionListener
{
   public ColorAction(Color c)
   {
      backgroundColor = c;
   }

   public void actionPerformed(ActionEvent event)
   {
      // set panel background color
      . . .
   }

   private Color backgroundColor;
}
```

然后，为每种颜色构造一个对象，并将这些对象设置为按钮监听器。

```
ColorAction yellowAction = new ColorAction(Color.YELLOW);
ColorAction blueAction = new ColorAction(Color.BLUE);
ColorAction redAction = new ColorAction(Color.RED);

yellowButton.addActionListener(yellowAction);
blueButton.addActionListener(blueAction);
redButton.addActionListener(redAction);
```

例如，如果一个用户在标有“Yellow”的按钮上点击了一下，yellowAction对象的actionPerformed方法就会被调用。这个对象的backgroundColor实例域被设置为Color.YELLOW，现在就将面板的背景色设置为黄色了。

这里还有一个需要考虑的问题。ColorAction对象不能访问buttonpanel变量。可以采用两种方式解决这个问题。一个是将面板存储在ColorAction对象中，并在ColorAction的构造器中设置

它；另一个是将ColorAction作为ButtonFrame类的内部类，这样一来，它的方法就自动地拥有访问外部面板的权限了（有关内部类的详细介绍请参看第6章）。

这里使用第二种方法。下面说明一下如何将ColorAction类放置在ButtonFrame类内。

```
class ButtonPanel extends JFrame
{
   . . .

   private class ColorAction implements ActionListener
   {
      . . .

      public void actionPerformed(ActionEvent event)
      {
         buttonPanel.setBackground(backgroundColor);
      }

      private Color backgroundColor;
   }
   private JPanel buttonPanel;
}
```

下面仔细地研究一下actionPerformed方法。在ColorAction类中没有buttonPanel 域，但在外部ButtonFrame类中却有。

这种情形经常会遇到。事件监听器对象通常需要执行一些对其他对象可能产生影响的操作。可以策略性地将监听器类放置在需要修改状态的那个类中。

例8-1包含了完整的程序。无论何时点击任何一个按钮，对应的动作监听器就会修改面板的背景颜色。

例8-1 ButtonTest .java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * @version 1.33 2007-06-12
 * @author Cay Horstmann
 */
public class ButtonTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               ButtonFrame frame = new ButtonFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
```

```
}

/**
 * A frame with a button panel
 */
class ButtonFrame extends JFrame
{
   public ButtonFrame()
   {
      setTitle("ButtonTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // create buttons
      JButton yellowButton = new JButton("Yellow");
      JButton blueButton = new JButton("Blue");
      JButton redButton = new JButton("Red");

      buttonPanel = new JPanel();

      // add buttons to panel
      buttonPanel.add(yellowButton);
      buttonPanel.add(blueButton);
      buttonPanel.add(redButton);

      // add panel to frame
      add(buttonPanel);

      // create button actions
      ColorAction yellowAction = new ColorAction(Color.YELLOW);
      ColorAction blueAction = new ColorAction(Color.BLUE);
      ColorAction redAction = new ColorAction(Color.RED);

      // associate actions with buttons
      yellowButton.addActionListener(yellowAction);
      blueButton.addActionListener(blueAction);
      redButton.addActionListener(redAction);
   }

   /**
    * An action listener that sets the panel's background color.
    */
   private class ColorAction implements ActionListener
   {
      public ColorAction(Color c)
      {
         backgroundColor = c;
      }

      public void actionPerformed(ActionEvent event)
      {
         buttonPanel.setBackground(backgroundColor);
      }

      private Color backgroundColor;
   }

   private JPanel buttonPanel;
```

```

    public static final int DEFAULT_WIDTH = 300;
    public static final int DEFAULT_HEIGHT = 200;
}
```

API javax.swing.JButton 1.2

- `JButton(String label)`
- `JButton(Icon icon)`
- `JButton(String label,Icon icon)`

  构造一个按钮。标签可以是常规的文本，从Java SE 1.3开始，也可以是HTML。例如，"<html>Ok</html>"。

API java.awt.Container 1.0

- `Component add(Component c)`

  将组件c添加到这个容器中。

API javax.swing.ImageIcon 1.2

- `ImageIcon(String filename)`

  构造一个图标，它的图像存储在一个文件中。

8.1.2 建议使用内部类

有些人不喜欢使用内部类，其原因是觉得类和对象的增殖会使得程序的执行速度变慢。下面讨论一下这个问题。首先，不需要为每个用户界面组件定义一个新类。在前面列举的示例中，三个按钮共享同一个监听器类。当然，每个按钮分别使用不同的监听器对象。但是，这些对象并不大，它们只包含一个颜色值和一个面板的引用。而使用传统的if. . .else语句的解决方案也需要引用动作监听器存储的上述颜色对象，只不过这是一个局部变量，而不是实例域。

下面是一个说明使用匿名内部类简化代码的示例。如果仔细看一下例8-1的代码，就会注意到每个按钮的处理过程都是一样的：

1）用标签字符串构造按钮。

2）将按钮添加到面板上。

3）用对应的颜色构造一个动作监听器。

4）添加动作监听器。

为了简化这些操作过程，可以设计一个辅助方法：

```
public void makeButton(String name, Color backgroundColor)
{
   JButton button = new JButton(name);
   buttonPanel.add(button);
   ColorAction action = new ColorAction(backgroundColor);
   button.addActionListener(action);
}
```

然后简化调用

```
makeButton("yellow", Color.YELLOW);
makeButton("blue", Color.BLUE);
makeButton("red", Color.RED);
```

接着，进一步进行简化。请注意，ColorAction类只在makeButton方法中使用一次。因此，可以将它设计为一个匿名类：

```
public void makeButton(String name, final Color backgroundColor)
{
   JButton button = new JButton(name);
   buttonPanel.add(button);
   button.addActionListener(new ActionListener()
      {
         public void actionPerformed(ActionEvent event)
         {
            buttonPanel.setBackground(backgroundColor);
         }
      });
}
```

动作监听器代码现在变得更加简单了。actionPerformed方法仅仅引用参数变量backgroundColor（与内部类中访问的所有局部变量一样，应该将参数声明为final）。

这里不需要显式的构造器。在第6章中已经看到，内部类机制将自动地生成一个构造器，其中存储着所有用在内部类方法中的final局部变量。

提示：匿名内部类看起来可能让人感觉有些困惑。如果训练自己的眼睛能够捕获程序代码中的关键字，那就可以破解它们，例如：

```
button.addActionListener(new ActionListener()
   {
      public void actionPerformed(ActionEvent event)
      {
         buttonPanel.setBackground(backgroundColor);
      }
   });
```

这就是说，按钮动作设置背景颜色。只要事件处理器包含的语句条数不多，就认为这段代码的可读性还是不错的，尤其是在对内部类机制没有什么抵触心理的情况下。

注释：任何实现了ActionListener接口的类对象都可以作为按钮监听器。我们更加倾向于为将要执行的按钮动作创建一个新类和类对象。然而，有些程序员不愿意使用内部类，却选择了不同的策略。他们找出因事件而改变的容器，让这些容器实现ActionListener接口。然后，将容器设置为监听器，如下所示：

```
yellowButton.addActionListener(this);
blueButton.addActionListener(this);
redButton.addActionListener(this);
```

注意，现在这里的三个按钮不再是独立的监听器。它们共享一个监听器对象，即按钮面板。因此，actionPerformed方法必须判断点击了哪个按钮。

```
class ButtonFrame extends JFrame implements ActionListener
```

```
    {
       . . .
       public void actionPerformed(ActionEvent event)
       {
          Object source = event.getSource();
          if (source == yellowButton) . . .
          else if (source == blueButton) . . .
          else if (source == redButton ) . . .
          else . . .
       }
    }
```

可以看到，这样会变得有些零散杂乱，我们不建议这样做。

API java.util.EventObject 1.1

- `Object getSource()`

  返回发生事件的对象引用。

API java.awt.event.ActionEvent 1.1

- `String getActionCommand()`

  返回与这个动作事件相关的命令字符串。如果动作事件来源于按钮，命令字符串就等于按钮标签，除非已经使用setActionCommand方法对字符串进行了修改。

API java.beans.EventHandler 1.4

- `static Object create(Class listenerInterface,Object target,String action)`
- `static Object create(Class listenerInterface, Object target, String action, String eventProperty)`
- `static Object create(Class listenerInterface, Object target, String action, String eventProperty, String listenerMethod)`

  构造实现给定接口的代理类对象。无论是命名方法，还是接口的所有方法都将执行目标对象上的给定动作。

  动作可以是一个方法名或目标的属性。如果是属性，则执行它的设置方法。例如，动作text将变为调用setText方法。

  事件属性由一个或多个用逗号分隔的属性名组成。第一个属性从监听器方法的参数中读出，第二个属性由结果对象读出，等等。最后的结果将作为动作的参数。例如，属性source.text将变为调用getSource和getText方法。

8.1.3 创建包含一个方法调用的监听器

在Java SE 1.4中引入了不用内部类来定义简单的事件监听器的机制。例如，假设有一个标签为Load的按钮，它的事件处理只包含下面一个方法调用：

```
frame.loadData();
```

当然，可以使用匿名内部类：

```
loadButton.addActionListener(new ActionListener()
   {
      public void actionPerformed(ActionEvent event)
      {
         frame.loadData();
      }
   });
```

但是，EventHandler类可以使用下列调用，自动地创建这样一个监听器：

```
EventHandler.create(ActionListener.class, frame, "loadData")
```

当然，仍然需要安装处理器：

```
loadButton.addActionListener(
   EventHandler.create(ActionListener.class, frame, "loadData"));
```

如果事件监听器调用的方法只包含一个从事件处理器继承来的参数，就可以使用另外一种形式的create方法。例如，调用

```
EventHandler.create(ActionListener.class, frame, "loadData", "source.text")
```

等价于

```
new ActionListener()
   {
      public void actionPerformed(ActionEvent event)
      {
         frame.loadData(((JTextField) event.getSource()).getText());
      }
   }
```

需要将属性source和text的名字转换成方法调用getSource和getText。

8.1.4 实例：改变观感

在默认情况下，Swing程序使用Metal观感，可以采用两种方式改变观感。第一种方式是在Java安装的子目录jre/lib下有一个文件swing.properties。在这个文件中，将属性swing.defaultlaf设置为所希望的观感类名。例如，

```
swing.defaultlaf=com.sun.java.swing.plaf.motif.MotifLookAndFeel
```

注意，Metal观感位于javax.swing包中。其他的观感包位于com.sun.java包中，并且不是在每个Java实现中都提供。现在，鉴于版权的原因，Windows和Mac的观感包只与Windows和Mac版本的Java运行时环境一起发布。

提示：由于属性文件中以#字符开始的行被忽略，所以，可以在swing.properties文件中提供几种观感选择，并通过增删#字符来切换选择：

```
#swing.defaultlaf=javax.swing.plaf.metal.MetalLookAndFeel
swing.defaultlaf=com.sun.java.swing.plaf.motif.MotifLookAndFeel
#swing.defaultlaf=com.sun.java.swing.plaf.windows.WindowsLookAndFeel
```

采用这种方式开启观感时必须重新启动程序。Swing程序只在启动时读取一次swing.properties文件。

第二种方式是动态地改变观感。这需要调用静态的UIManager.setLookAndFeel方法，并提供所想要的观感类名，然后再调用静态方法SwingUtilities.updateComponentTreeUI来刷新全部的组件集。这里需要向这个方法提供一个组件，并由此找到其他的所有组件。当UIManager.setLookAndFeel方法没有找到所希望的观感或在加载过程中出现错误时，将会抛出异常。与前面一样，建议暂且将异常处理的代码跳过，等到第11章详细地讲述异常时就会理解了。

下面是一个示例，它显示了如何用程序切换至Motif观感：

```
String plaf = "com.sun.java.swing.plaf.motif.MotifLookAndFeel";
try
{
   UIManager.setLookAndFeel(plaf);
   SwingUtilities.updateComponentTreeUI(panel);
}
catch(Exception e) { e.printStackTrace(); }
```

为了列举安装的所有观感实现，可以调用

```
UIManager.LookAndFeelInfo[] infos = UIManager.getInstalledLookAndFeels();
```

然后采用下列方式得到每一种观感的名字和类名

```
String name = infos[i].getName();
String className = infos[i].getClassName();
```

例8-2是一个完整的程序，它演示了如何切换观感（如图8-4所示）的方式。这个程序与例8-1十分相似。我们遵循前一节的建议，使用辅助方法makeButton和匿名内部类指定按钮动作，即切换观感。

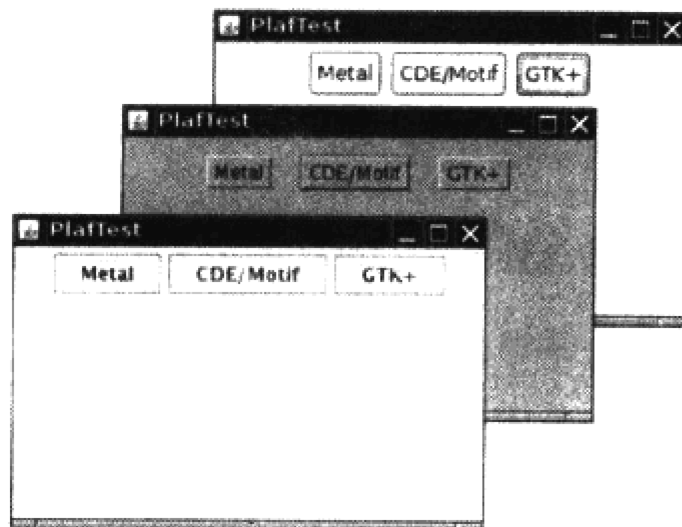


图8-4　切换观感

在这个程序中，还有一点需要注意的地方。在内部动作监听器类的actionPerformed方法中，需要将一个外部PlafFrame类的this引用传递给updateComponentTreeUI方法。回想一下第6章所说过的，外部对象的this指针必须将外部类名作为前缀：

```
SwingUtilities.updateComponentTreeUI(PlafPanel.this);
```

例8-2　PlafTest.java

```
import java.awt.EventQueue;
import java.awt.event.*;
import javax.swing.*;

/**
 * @version 1.32 2007-06-12
 * @author Cay Horstmann
 */
public class PlafTest
{
   public static void main(String[] args)
   {
```

```
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               PlafFrame frame = new PlafFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * A frame with a button panel for changing look and feel
 */
class PlafFrame extends JFrame
{
   public PlafFrame()
   {
      setTitle("PlafTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      buttonPanel = new JPanel();

      UIManager.LookAndFeelInfo[] infos = UIManager.getInstalledLookAndFeels();
      for (UIManager.LookAndFeelInfo info : infos)
         makeButton(info.getName(), info.getClassName());

      add(buttonPanel);
   }

   /**
    * Makes a button to change the pluggable look and feel.
    * @param name the button name
    * @param plafName the name of the look and feel class
    */
   void makeButton(String name, final String plafName)
   {
      // add button to panel

      JButton button = new JButton(name);
      buttonPanel.add(button);

      // set button action

      button.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               // button action: switch to the new look and feel
               try
               {
                  UIManager.setLookAndFeel(plafName);
                  SwingUtilities.updateComponentTreeUI(PlafFrame.this);
               }
               catch (Exception e)
               {
```

```
                e.printStackTrace();
             }
          }
       });
   }

   private JPanel buttonPanel;

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;
}
```

API javax.swing.UIManager 1.2

- `static UIManager.LookAndFeelInfo[] getInstalledLookAndFeels()`

  获得一个用于描述已安装的观感实现的对象数组。

- `static setLookAndFeel(String className)`

  利用给定的类名设置当前的观感。例如，javax.swing.plaf.metal.MetalLookAndFeel

API javax.swing.UIManager.LookAndFeelInfo 1.2

- `String getName()`

  返回观感的显示名称。

- `String getClassName()`

  返回观感实现类的名称。

8.1.5 适配器类

并不是所有的事件处理都像按钮点击那样简单。在正规的程序中，往往希望用户在确认没有丢失所做工作之后再关闭程序。当用户关闭框架时，可能希望弹出一个对话框来警告用户没有保存的工作有可能会丢失，只有在用户确认之后才退出程序。

当程序用户试图关闭一个框架窗口时，JFrame对象就是WindowEvent的事件源。如果希望捕获这个事件，就必须有一个合适的监听器对象，并将它添加到框架的窗口监听器列表中。

```
WindowListener listener = . . .;
frame.addWindowListener(listener);
```

窗口监听器必须是实现WindowListener接口的类的一个对象。在WindowListener接口中包含7个方法。当发生窗口事件时，框架将调用这些方法响应7个不同的事件。从它们的名字就可以得知其作用，惟一的例外是在Windows下，通常将iconified称为minimized。下面是完整的WindowListener接口：

```
public interface WindowListener
{
   void windowOpened(WindowEvent e);
   void windowClosing(WindowEvent e);
   void windowClosed(WindowEvent e);
   void windowIconified(WindowEvent e);
   void windowDeiconified(WindowEvent e);
   void windowActivated(WindowEvent e);
```

```
    void windowDeactivated(WindowEvent e);
}
```

注释：为了能够查看窗口是否被最大化，需要安装WindowStateListener。有关更加详细的信息请参看后面的API注释。

正像前面曾经说过的那样，在Java中，实现一个接口的任何类都必须实现其中的所有方法；在这里，意味着需要实现7个方法。然而只对名为windowClosing的方法感兴趣。

当然，可以这样定义实现这个接口的类：在windowClosing方法中增加一个对System.exit(0) 的调用，其他6个方法不做任何事情：

```
class Terminator implements WindowListener
{
   public void windowClosing(WindowEvent e)
   {
      if (user agrees)
         System.exit(0);
   }

   public void windowOpened(WindowEvent e) {}
   public void windowClosed(WindowEvent e) {}
   public void windowIconified(WindowEvent e) {}
   public void windowDeiconified(WindowEvent e) {}
   public void windowActivated(WindowEvent e) {}
   public void windowDeactivated(WindowEvent e) {}
}
```

书写6个没有任何操作的方法代码显然是一种乏味的工作。鉴于简化的目的，每个含有多个方法的AWT监听器接口都配有一个*适配器*（adapter）类，这个类实现了接口中的所有方法，但每个方法没有做任何事情。这意味着适配器类自动地满足了Java实现相关监听器接口的技术需求。可以通过扩展适配器类来指定对某些事件的响应动作，而不必实现接口中的每个方法（ActionListener这样的接口只有一个方法，因此没必要提供适配器类）。

下面使用窗口适配器。首先定义一个WindowAdapter类的扩展类，其中包含继承的6个没有做任何事情的方法和一个覆盖的方法windowClosing：

```
class Terminator extends WindowAdapter
{
   public void windowClosing(WindowEvent e)
   {
      if (user agrees)
         System.exit(0);
   }
}
```

现在，可以将一个Terminator对象注册为事件监听器：

```
WindowListener listener = new Terminator();
frame.addWindowListener(listener);
```

只要框架产生了窗口事件，就会通过调用7个方法之中的一个方法将事件传递给listener对象（如图8-5所示），其中6个方法没有做任何事情；windowClosing方法调用System.exit(0)终止应用程序的执行。

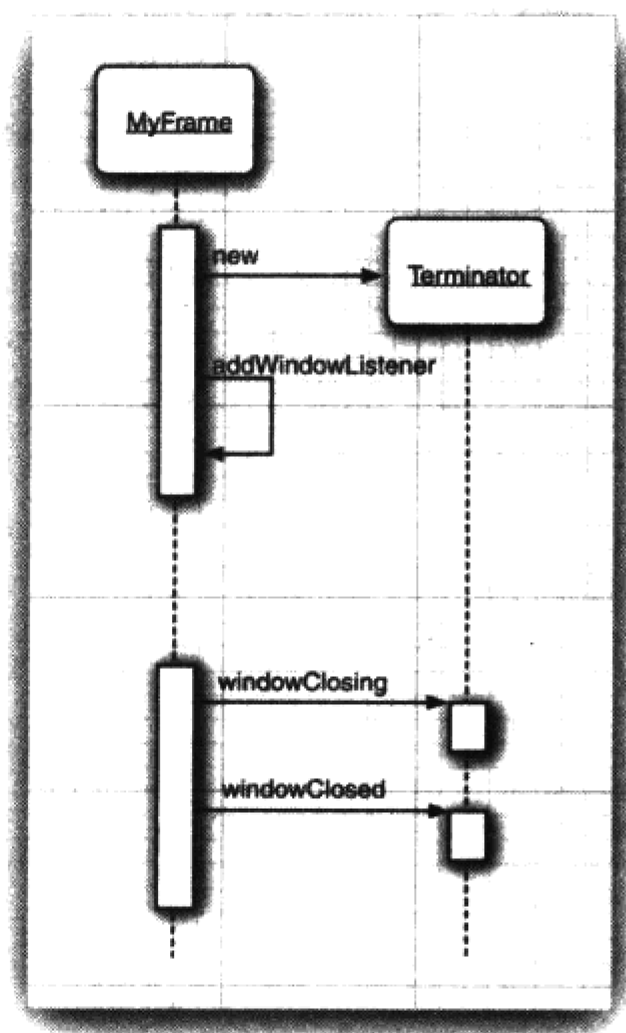


图8-5 窗口监听器

警告：如果在扩展适配器类时将方法名拼写错了，编译器不会捕获到这个错误。例如，如果在WindowAdapter类中定义一个windowIsClosing方法，就会得到一个包含8个方法的类，并且windowClosing方法没有做任何事情。

创建一个扩展于WindowAdapter的监听器类是一个很好的改进，但是还可以继续改进。事实上，没有必要为listener对象命名。只需写成：

```
frame.addWindowListener(new Terminator());
```

不要就此止步！我们可以将监听器类定义为框架的匿名内部类。

```
frame.addWindowListener(new
   WindowAdapter()
   {
      public void windowClosing(WindowEvent e)
      {
         if (user agrees)
            System.exit(0);
      }
   });
```

这段代码具有下列作用：

- 定义了一个扩展于WindowAdapter类的无名类。
- 将windowClosing方法添加到匿名类中（与前面一样，这个方法将退出程序）。
- 从WindowAdapter继承6个没有做任何事情的方法。

- 创建这个类的一个对象，这个对象没有名字。
- 将这个对象传递给addWindowListener方法。

这里再次说明一下，匿名内部类的语法需要人们适应一段时间，但得到的是更加简练的代码。

API java.awt.event.WindowListener 1.1

- `void windowOpened(WindowEvent e)`
  窗口打开后调用这个方法。
- `void windowClosing(WindowEvent e)`
  在用户发出窗口管理器命令关闭窗口时调用这个方法。需要注意一点，仅当调用hide或dispose方法后窗口才能够关闭。
- `void windowClosed(WindowEvent e)`
  窗口关闭后调用这个方法。
- `void windowIconified(WindowEvent e)`
  窗口图标化后调用这个方法。
- `void windowDeiconified(WindowEvent e)`
  窗口非图标化后调用这个方法。
- `void windowActivated(WindowEvent e)`
  激活窗口后调用这个方法。只有框架或对话框可以被激活。通常，窗口管理器会对活动窗口进行修饰，比如，高亮度标题栏。
- `void windowDeactivated(WindowEvent e)`
  窗口变为未激活状态后调用这个方法。

API java.awt.event.WindowStateListener 1.4

- `void windowStateChanged(WindowEvent event)`
  窗口被极大化、图标化或恢复为正常大小时调用这个方法。

API java.awt.event.WindowEvent 1.1

- `int getNewState() 1.4`
- `int getOldState() 1.4`
  返回窗口状态改变事件中窗口的新、旧状态。返回的整型数值是下列数值之一：

```
Frame.NORMAL
Frame.ICONIFIED
Frame.MAXIMIZED_HORIZ
Frame.MAXIMIZED_VERT
Frame.MAXIMIZED_BOTH
```

8.2 动作

通常，激活一个命令可以有多种方式。用户可以通过菜单、击键或工具栏上的按钮选择特定的功能。在AWT事件模型中实现这些非常容易：将所有事件连接到同一个监听器上。例如，假设blueAction是一个动作监听器，它的actionPerformed方法可以将背景颜色改变成蓝色。将一个监听器对象加到下面几个事件源上：

- 标记为Blue的工具栏按钮
- 标记为Blue的菜单项
- 击键CTRL+B

然后，无论改变背景颜色的命令是通过哪种方式下达的，是点击按钮、菜单选择，还是按下键盘，其操作动作都是一样的。

Swing包提供了一种非常实用的机制来封装命令，并将它们连接到多个事件源，这就是Action接口。一个动作是一个封装下列内容的对象：

- 命令的说明（一个文本字符串和一个可选图标）；
- 执行命令所需要的参数（例如，在列举的例子中请求改变的颜色）。

Aceron接口包含下列方法

```
void actionPerformed(ActionEvent event)
void setEnabled(boolean b)
boolean isEnabled()
void putValue(String key, Object value)
Object getValue(String key)
void addPropertyChangeListener(PropertyChangeListener listener)
void removePropertyChangeListener(PropertyChangeListener listener)
```

第一个方法是ActionListener接口中很熟悉的一个：实际上，Action接口扩展于Action Listener接口，因此，可以在任何需要ActionListener对象的地方使用Action对象。

接下来的两个方法允许启用或禁用这个动作，并检查这个动作当前是否启用。当一个连接到菜单或工具栏上的动作被禁用时，这个选项就会变成灰色。

putValue和getvalue方法允许存储和检索动作对象中的任意名/值。有两个重要的预定义字符串：Action.NAME和Action.SMALL_ICON，用于将动作的名字和图标存储到一个动作对象中：

```
action.putValue(Action.NAME, "Blue");
action.putValue(Action.SMALL_ICON, new ImageIcon("blue-ball.gif"));
```

表8-1给出了所有预定义的动作表名称。

表8-1 预定义动作表名称

名 称	值
NAME	动作名称，显示在按钮和菜单上
SMALL_ICON	存储小图标的地方；显示在按钮、菜单项或工具栏中
SHORT_DESCRIPTION	图标的简要说明；显示在工具提示中
LONG_DESCRIPTION	图标的详细说明；使用在在线帮助中。没有Swing组件使用这个值
MNEMONIC_KEY	快捷键缩写；显示在菜单项中（请参看第9章）
ACCELERATOR_KEY	存储加速击键的地方；Swing组件不使用这个值
ACTION_COMMAND_KEY	历史遗留；仅在旧版本的registerKeyboardAction方法中使用
DEFAULT	常用的综合属性；Swing组件不使用这个值

如果动作对象添加到菜单或工具栏上，它的名称和图标就会被自动地提取出来，并显示在菜单项或工具栏项中。SHORT_DESCRIPTION值变成了工具提示。

Action接口的最后两个方法能够让其他对象在动作对象的属性发生变化时得到通告，尤其是菜单或工具栏触发的动作。例如，如果增加一个菜单，作为动作对象的属性变更监听器，而这个动作对象随后被禁用，菜单就会被调用，并将动作名称变为灰色。属性变更监听器是一种常用的构造形式，它是JavaBeans组件模型的一部分。有关这方面更加详细的信息请参看卷II。

需要注意，Action是一个*接口*，而不是一个类。实现这个接口的所有类都必须实现刚才讨论的7个方法。庆幸的是，有一个类实现了这个接口除actionPerformed方法之外的所有方法，它就是AbstractAction。这个类存储了所有名/值对，并管理着属性变更监听器。我们可以直接扩展AbstractAction类，并在扩展类中实现actionPerformed方法。

下面构造一个用于执行改变颜色命令的动作对象。首先存储这个命令的名称、图标和需要的颜色。将颜色存储在AsbstractAction类提供的名/值对表中。下面是ColorAction类的代码。构造器设置名/值对，而actionPerformed方法执行改变颜色的动作。

```
public class ColorAction extends AbstractAction
{
   public ColorAction(String name, Icon icon, Color c)
   {
      putValue(Action.NAME, name);
      putValue(Action.SMALL_ICON, icon);
      putValue("color", c);
      putValue(Action.SHORT_DESCRIPTION, "Set panel color to " + name.toLowerCase());
   }

   public void actionPerformed(ActionEvent event)
   {
      Color c = (Color) getValue("color");
      buttonPanel.setBackground(c);
   }
}
```

在测试程序中，创建了三个这个类的对象，如下所示：

```
Action blueAction = new ColorAction("Blue", new ImageIcon("blue-ball.gif"), Color.BLUE);
```

接下来，将这个动作与一个按钮关联起来。由于JButton有一个用Action对象作为参数的构造器，所以实现这项操作很容易。

```
JButton blueButton = new JButton(blueAction);
```

构造器读取动作的名称和图标，为工具提示设置简要说明，将动作设置为监听器。从图8-6中可以看到图标和工具提示。在下一章中将会看到，将这个动作添加到菜单中也非常容易。

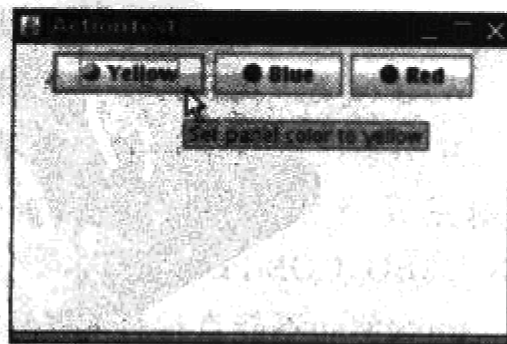


图8-6 按钮显示来自动作对象的图标

最后，想要将这个动作对象添加到击键中，以便让用户敲击键盘命令来执行这项动作。为了将动作与击键关联起来，首先需要生成KeyStroke类对象。这是一个很有用的类，它封装了对键的说明。要想生成一个KeyStroke对象，

不要调用构造器，而是调用KeyStroke类中的静态getKeyStroke方法：

```
KeyStroke ctrlBKey = KeyStroke.getKeyStroke("ctrl B");
```

为了能够理解下一个步骤，需要知道*keyboard focus*的概念。用户界面中可以包含许多按钮、菜单、滚动栏以及其他的组件。当用户敲击键盘时，这个动作会被发送给拥有焦点的组件。通常具有焦点的组件可以明显地察觉到（但并不总是这样），例如，在Java观感中，具有焦点的按钮在按钮文本周围有一个细的矩形边框。用户可以使用TAB键在组件之间移动焦点。当按下SPACE键时，就点击了拥有焦点的按钮。还有一些键执行一些其他的动作，例如，按下箭头键可以移动滚动条。

然而，在这里的示例中，并不希望将击键发送给拥有焦点的组件。否则，每个按钮都需要知道如何处理CTRL+Y、CTRL+B 和CTRL+R这些组合键。

这是一个常见的问题，Swing设计者给出了一种很便捷的解决方案。每个JComponent有三个输入映射（imput maps），每一个映射的KeyStroke对象都与动作关联。三个输入映射对应着三个不同的条件（请参看表8-2）。

表8-2 输入映射条件

标　志	激活动作
WHEN_FOCUSED	当这个组件拥有键盘焦点时
WHEN_ANCESTOR_OF_FOCUSED_COMPONENT	当这个组件包含了拥有键盘焦点的组件时
WHEN_IN_FOCUSED_WINDOW	当这个组件被包含在一个拥有键盘焦点组件的窗口中时

击键处理将按照下列顺序检查这些映射：

1）检查具有输入焦点组件的WHEN_FOCUSED映射。如果这个击键存在，将执行对应的动作。如果动作已启用，则停止处理。

2）从具有输入焦点的组件开始，检查其父组件的WHEN_ANCESTOR_OF_FOCUSED_COMPONENT映射。一旦找到击键对应的映射，就执行对应的动作。如果动作已启用，将停止处理。

3）查看具有输入焦点的窗口中的所有可视的和启用的组件，这个击键被注册到WHEN_IN_FOCUSED_WINDOW映射中。给这些组件（按照击键注册的顺序）一个执行对应动作的机会。一旦第一个启用的动作被执行，就停止处理。如果一个击键在多个WHEN_IN_FOCUSED_WINDOW映射中出现，这部分处理就可能会出现问题。

可以使用getInputMap方法从组件中得到输入映射。例如：

```
InputMap imap = panel.getInputMap(JComponent.WHEN_FOCUSED);
```

WHEN_FOCUSED条件意味着在当前组件拥有键盘焦点时会查看这个映射。在这里的示例中，并不想使用这个映射。是某个按钮拥有输入焦点，而不是面板。其他的两个映射都能够很好地完成增加颜色改变击键的任务。在示例程序中使用的是WHEN_ANCESTOR_OF_FOCUSED_COMPONENT。

InputMap不能直接地将KeyStroke对象映射到Action对象。而是先映射到任意对象上，然后由ActionMap类实现将对象映射到动作上的第2个映射。这样很容易实现来自不同输入映射的击

键共享一个动作的目的。

因而，每个组件都可以有三个输入映射和一个动作映射。为了将它们组合起来，需要为动作命名。下面是将键与动作关联起来的方式：

```
imap.put(KeyStroke.getKeyStroke("ctrl Y"), "panel.yellow");
ActionMap amap = panel.getActionMap();
amap.put("panel.yellow", yellowAction);
```

习惯上，使用字符串none表示空动作。这样可以轻松地取消一个键动作：

```
imap.put(KeyStroke.getKeyStroke("ctrl C"), "none");
```

警告：JDK文档提倡使用动作名作为动作键。我们并不认为这是一个好建议。在按钮和菜单项上显示的动作名，UI设计者可以随心所欲地进行更改，也可以将其翻译成多种语言。使用这种不牢靠的字符串作为查询键不是一种好的选择。建议将动作名与显示的名字分开。

下面总结一下用同一个动作响应按钮、菜单项或击键的方式：

1）实现一个扩展于AbstractAction类的类。多个相关的动作可以使用同一个类。

2）构造一个动作类的对象。

3）使用动作对象创建按钮或菜单项。构造器将从动作对象中读取标签文本和图标。

4）为了能够通过击键触发动作，必须额外地执行几步操作。首先定位顶层窗口组件，例如，包含所有其他组件的面板。

5）然后，得到顶层组件的WHEN_ANCESTOR_OF_FOCUS_COMPONENT输入映射。为需要的击键创建一个KeyStrike对象。创建一个描述动作字符串这样的动作键对象。将（击键，动作键）对添加到输入映射中。

6）最后，得到顶层组件的动作映射。将（动作键，动作对象）添加到映射中。

例8-3给出了将按钮和击键映射到动作对象的完整程序代码。试试看，点击按钮或按下CTRL+Y、CTRL+B或CTRL+R来改变面板颜色。

例8-3 ActionTest.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * @version 1.33 2007-06-12
 * @author Cay Horstmann
 */
public class ActionTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
```

```
                ActionFrame frame = new ActionFrame();
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
             }
          });
    }
}

/**
 * A frame with a panel that demonstrates color change actions.
 */
class ActionFrame extends JFrame
{
   public ActionFrame()
   {
      setTitle("ActionTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      buttonPanel = new JPanel();

      // define actions
      Action yellowAction = new ColorAction("Yellow", new ImageIcon("yellow-ball.gif"),
            Color.YELLOW);
      Action blueAction = new ColorAction("Blue", new ImageIcon("blue-ball.gif"), Color.BLUE);
      Action redAction = new ColorAction("Red", new ImageIcon("red-ball.gif"), Color.RED);

      // add buttons for these actions
      buttonPanel.add(new JButton(yellowAction));
      buttonPanel.add(new JButton(blueAction));
      buttonPanel.add(new JButton(redAction));

      // add panel to frame
      add(buttonPanel);

      // associate the Y, B, and R keys with names
      InputMap imap = buttonPanel.getInputMap(JComponent.WHEN_ANCESTOR_OF_FOCUSED_COMPONENT);
      imap.put(KeyStroke.getKeyStroke("ctrl Y"), "panel.yellow");
      imap.put(KeyStroke.getKeyStroke("ctrl B"), "panel.blue");
      imap.put(KeyStroke.getKeyStroke("ctrl R"), "panel.red");

      // associate the names with actions
      ActionMap amap = buttonPanel.getActionMap();
      amap.put("panel.yellow", yellowAction);
      amap.put("panel.blue", blueAction);
      amap.put("panel.red", redAction);
   }

   public class ColorAction extends AbstractAction
   {
      /**
       * Constructs a color action.
       * @param name the name to show on the button
       * @param icon the icon to display on the button
       * @param c the background color
       */
      public ColorAction(String name, Icon icon, Color c)
      {
```

```
            putValue(Action.NAME, name);
            putValue(Action.SMALL_ICON, icon);
            putValue(Action.SHORT_DESCRIPTION, "Set panel color to " + name.toLowerCase());
            putValue("color", c);
         }

         public void actionPerformed(ActionEvent event)
         {
            Color c = (Color) getValue("color");
            buttonPanel.setBackground(c);
         }
      }

      private JPanel buttonPanel;

      public static final int DEFAULT_WIDTH = 300;
      public static final int DEFAULT_HEIGHT = 200;
   }
```

API javax.swing.Action 1.2

- `boolean isEnalbled()`
- `void setEnabled(boolean b)`

  获得或设置这个动作的enabled属性。

- `void putValue(String key,Object value)`

  将名/值对放置在动作对象内。

  参数：key　　用动作对象存储性能的名字。它可以是一个字符串，但预定义了几个名字，其含义参看表8-1。

  value　　与名字关联的对象。

- `Object getValue(String key)`

  返回被存储的名/值对的值。

API javax.swing.KeyStroke 1.2

- `static KeyStroke getKeyStroke(String description)`

  根据一个便于人们阅读的说明创建一个击键(由空格分隔的字符串序列)。这个说明以0个或多个修饰符 shift control ctrl meta alt altGraph开始，以由typed和单个字符构成的字符串（例如：“typed a”）或者一个可选的事件说明符（pressed默认，或released）紧跟一个键码结束。以VK_前缀开始的键码应该对应一个KeyEvent常量，例如，“INSERT”对应KeyEvent.VK_INSERT。

API javax.swing.JComponent 1.2

- `ActionMap getActionMap()`　1.3

  返回关联动作映射键（可以是任意的对象）和动作对象的映射。

- `InputMap getInputMap(int flag)`　1.3

获得将击键映射到动作键的输入映射。

参数：flag 触发动作的键盘焦点条件。具体的值请参看表8-2。

8.3 鼠标事件

如果只希望用户能够点击按钮或菜单，那么就不需要显式地处理鼠标事件。鼠标操作将由用户界面中的各种组件内部处理。然而，如果希望用户使用鼠标画图，就需要捕获鼠标移动点击和拖动事件。

在本节中，将展示一个简单的图形编辑器应用程序，它允许用户在画布上（如图8-7所示）放置、移动和擦除方块。

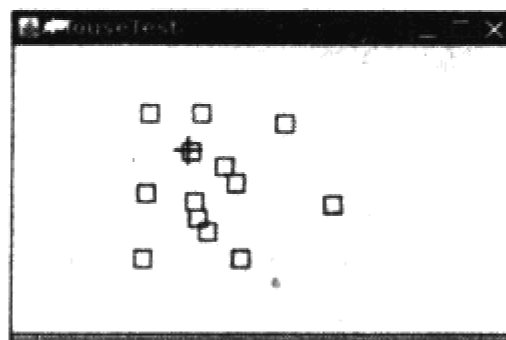


图8-7 鼠标测试程序

当用户点击鼠标按钮时，将会调用三个监听器方法：鼠标第一次被按下时调用mousePressed；鼠标被释放时调用mouseReleased；最后调用mouseClicked。如果只对最终的点击事件感兴趣，就可以忽略前两个方法。用MouseEvent类对象作为参数，调用getX和getY方法可以获得鼠标被按下时鼠标指针所在的x和y坐标。要想区分单击、双击和三击（!），需要使用getClickCount方法。

有些用户界面设计者喜欢让用户采用鼠标点击与键盘修饰符组合（例如，CONTROL+SHIFT+CLICK）的方式进行操作。我们感觉这并不是一种值得赞许的方式。如果对此持有不同的观点，可以看一看同时检测鼠标按键和键盘修饰符所带来的混乱。

可以采用位掩码来测试已经设置了哪个修饰符。在最初的API中，有两个按钮的掩码与两个键盘修饰符的掩码一样，即

```
BUTTON2_MASK == ALT_MASK
BUTTON3_MASK == META_MASK
```

这样做是为了能够让用户使用仅有一个按钮的鼠标通过按下修饰符键来模拟按下其他鼠标键的操作。然而，在Java SE 1.4中，建议使用一种不同的方式。有下列掩码：

```
BUTTON1_DOWN_MASK
BUTTON2_DOWN_MASK
BUTTON3_DOWN_MASK
SHIFT_DOWN_MASK
CTRL_DOWN_MASK
ALT_DOWN_MASK
ALT_GRAPH_DOWN_MASK
META_DOWN_MASK
```

getModifiersEx方法能够准确地报告鼠标事件的鼠标按钮和键盘修饰符。

需要注意，在Windows环境下，使用BUTTON3_DOWN_MASK检测鼠标右键（非主要的）的状态。例如，可以使用下列代码检测鼠标右键是否被按下：

```
if ((event.getModifiersEx() & InputEvent.BUTTON3_DOWN_MASK) != 0)
   . . . // code for right click
```

在列举的简单示例中，提供了mousePressed和mouseClicked方法。当鼠标点击在所有小方块的像素之外时，就会绘制一个新的小方块。这个操作是在mousePressed方法中实现的，这样可以让用户的操作立即得到响应，而不必等到释放鼠标按键。如果用户在某个小方块中双击鼠

标，就会将它擦除。由于需要知道点击次数，所以这个操作将在mouseClick方法中实现。

```
public void mousePressed(MouseEvent event)
{
   current = find(event.getPoint());
   if (current == null) // not inside a square
      add(event.getPoint());
}

public void mouseClicked(MouseEvent event)
{
   current = find(event.getPoint());
   if (current != null && event.getClickCount() >= 2)
      remove(current);
}
```

当鼠标在窗口上移动时，窗口将会收到一连串的鼠标移动事件。请注意：有两个独立的接口MouseListener和MouseMotionListener。这样做有利于提高效率。当用户移动鼠标时，只关心鼠标*点击*（*clicks*）的监听器就不会被多余的鼠标*移动*（*moves*）所困扰。

这里给出的测试程序将捕获鼠标动作事件，以便在光标位于一个小方块之上时变成另外一种形状（十字）。实现这项操作需要使用Cursor类中的getPredefinedCursor方法。表8-3列出了在Windows环境下，鼠标的形状和方法对应的常量（注意，有若干个光标的形状完全一样，但在其他平台上未必如此）。

表8-3　光标形状样例

图　标	常　量	图　标	常　量
	DEFAULT_CURSOR		NE_RESIZE_CURSOR
	CROSSHAIR_CURSOR		E_RESIZE_CURSOR
	HAND_CURSOR		SE_RESIZE_CURSOR
	MOVE_CURSOR		S_RESIZE_CURSOR
	TEXT_CURSOR		SW_RESIZE_CURSOR
	WAIT_CURSOR		W_RESIZE_CURSOR
	N_RESIZE_CURSOR		NW_RESIZE_CURSOR

下面是示例程序中MouseMotionListener类的mouseMoved方法：

```
public void mouseMoved(MouseEvent event)
{
   if (find(event.getPoint()) == null)
      setCursor(Cursor.getDefaultCursor());
   else
      setCursor(Cursor.getPredefinedCursor(Cursor.CROSSHAIR_CURSOR));
}
```

注释：可以利用Toolkit类中的createCustomCursor方法自定义光标类型：

```
Toolkit tk = Toolkit.getDefaultToolkit();
Image img = tk.getImage("dynamite.gif");
Cursor dynamiteCursor = tk.createCustomCursor(img, new Point(10, 10), "dynamite stick");
```

createCustomCursor的第一个参数指向光标图像。第二个参数给出了光标的“热点”偏移。第三个参数是一个描述光标的字符串。这个字符串可以用于访问性支持，例如，可以将光标形式读给视力受损或没有在屏幕前面的人。

如果用户在移动鼠标的同时按下鼠标，就会调用mouseMoved而不是调用mouseDragged。在测试应用程序中，用户可以用光标拖动小方块。在程序中，仅仅用拖动的矩形更新当前光标位置。然后，重新绘制画布，以显示新的鼠标位置。

```
public void mouseDragged(MouseEvent event)
{
   if (current != null)
   {
      int x = event.getX();
      int y = event.getY();

      current.setFrame(x - SIDELENGTH / 2, y - SIDELENGTH / 2, SIDELENGTH, SIDELENGTH);
      repaint();

}
```

注释：只有鼠标在一个组件内部停留才会调用mouseMoved方法。然而，即使鼠标拖动到组件外面，mouseDragged方法也会被调用。

还有两个鼠标事件方法：mouseEntered和mouseExited。这两个方法是在鼠标进入或移出组件时被调用。

最后，解释一下如何监听鼠标事件。鼠标点击由mouseClicked过程报告，它是MouseListener接口的一部分。由于大部分应用程序只对鼠标点击感兴趣，而对鼠标移动并不感兴趣，但鼠标移动事件发生的频率又很高，因此将鼠标移动事件与拖动事件定义在一个称为MouseMotionListener的独立接口中。

在示例程序中，对两种鼠标事件类型都感兴趣。这里定义了两个内部类：MouseHandler和MouseMotionHandler。MouseHandler类扩展于MouseAdapter类，这是因为它只定义了5个MouseListener方法中的两个方法。MouseMotionHandler实现了MouseMotionListener接口，并定义了这个接口中的两个方法。例8-4是这个程序的清单：

例8-4 MouseTest.java

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.awt.geom.*;
import javax.swing.*;

/**
 * @version 1.32 2007-06-12
```

```
 * @author Cay Horstmann
 */
public class MouseTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               MouseFrame frame = new MouseFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * A frame containing a panel for testing mouse operations
 */
class MouseFrame extends JFrame
{
   public MouseFrame()
   {
      setTitle("MouseTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // add component to frame

      MouseComponent component = new MouseComponent();
      add(component);
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;
}

/**
 * A component with mouse operations for adding and removing squares.
 */
class MouseComponent extends JComponent
{
   public MouseComponent()
   {
      squares = new ArrayList<Rectangle2D>();
      current = null;

      addMouseListener(new MouseHandler());
      addMouseMotionListener(new MouseMotionHandler());
   }

   public void paintComponent(Graphics g)
   {
      Graphics2D g2 = (Graphics2D) g;

      // draw all squares
```

```
            for (Rectangle2D r : squares)
               g2.draw(r);
         }

         /**
          * Finds the first square containing a point.
          * @param p a point
          * @return the first square that contains p
          */
         public Rectangle2D find(Point2D p)
         {
            for (Rectangle2D r : squares)
            {
               if (r.contains(p)) return r;
            }
            return null;
         }

         /**
          * Adds a square to the collection.
          * @param p the center of the square
          */
         public void add(Point2D p)
         {
            double x = p.getX();
            double y = p.getY();

            current = new Rectangle2D.Double(x - SIDELENGTH / 2, y - SIDELENGTH / 2, SIDELENGTH,
                  SIDELENGTH);
            squares.add(current);
            repaint();
         }

         /**
          * Removes a square from the collection.
          * @param s the square to remove
          */
         public void remove(Rectangle2D s)
         {
            if (s == null) return;
            if (s == current) current = null;
            squares.remove(s);
            repaint();
         }

         private static final int SIDELENGTH = 10;
         private ArrayList<Rectangle2D> squares;
         private Rectangle2D current;

         // the square containing the mouse cursor

         private class MouseHandler extends MouseAdapter
         {
            public void mousePressed(MouseEvent event)
            {
               // add a new square if the cursor isn't inside a square
               current = find(event.getPoint());
```

```java
            if (current == null) add(event.getPoint());
         }

         public void mouseClicked(MouseEvent event)
         {
            // remove the current square if double clicked
            current = find(event.getPoint());
            if (current != null && event.getClickCount() >= 2) remove(current);
         }
      }

      private class MouseMotionHandler implements MouseMotionListener
      {
         public void mouseMoved(MouseEvent event)
         {
            // set the mouse cursor to cross hairs if it is inside
            // a rectangle

            if (find(event.getPoint()) == null) setCursor(Cursor.getDefaultCursor());
            else setCursor(Cursor.getPredefinedCursor(Cursor.CROSSHAIR_CURSOR));
         }

         public void mouseDragged(MouseEvent event)
         {
            if (current != null)
            {
               int x = event.getX();
               int y = event.getY();

               // drag the current rectangle to center it at (x, y)
               current.setFrame(x - SIDELENGTH / 2, y - SIDELENGTH / 2, SIDELENGTH, SIDELENGTH);
               repaint();
            }
         }
      }
}
```

API java.awt.event.MouseEvent 1.1

- **int getX()**
- **int getY()**
- **Point getPoint()**

  返回事件发生时，事件源组件左上角的坐标x（水平）和y（竖直），或点信息。
- **int getClickCount()**

  返回与事件关联的鼠标连击次数（“连击”所指定的时间间隔与具体系统有关）。

API java.awt.event.InputEvent 1.1

- **int getModifiersEx() 1.4**

  返回事件扩展的或“按下”（down）的修饰符。使用下面的掩码值检测返回值：

  BUTTON1_DOWN_MASK

BUTTON2_DOWN_MASK
BUTTON3_DOWN_MASK
SHIFT_DOWN_MASK
CTRL_DOWN_MASK
ALT_DOWN_MASK
ALT_GRAPH_DOWN_MASK
META_DOWN_MASK

- `static String getModifiersExText(int modifiers) 1.4`
  返回用给定标志集描述的扩展或“按下”（down）的修饰符字符串，例如“Shift+Button1”。

API java.awt.Toolkit 1.0

- `public Cursor createCustomCursor(Image image,Point hotSpot,String name) 1.2`
  创建一个新的定制光标对象。
  参数：image　光标活动时显示的图像
  hotSpot　光标热点（箭头的顶点或十字中心）
  name　光标的描述，用来支持特殊的访问环境

API java.awt.Component 1.0

- `public void setCursor(Cursor cursor) 1.1`
  用光标图像设置给定光标。

8.4 AWT事件继承层次

弄清了事件处理的工作过程之后，作为本章的结束，总结一下AWT事件处理的体系架构。

前面已经提到，Java事件处理采用的是面向对象方法，所有的事件都是由java.util包中的EventObject类扩展而来的（公共超类不是Event，它是旧事件模型中的事件类名。尽管现在不赞成使用旧的事件模型，但这些类仍然保留在Java库中）。

EventObject类有一个子类AWTEvent，它是所有AWT事件类的父类。图8-8显示了AWT事件的继承关系图。

有些Swing组件将生成其他事件类型的事件对象；它们都直接扩展于EventObject，而不是AWTEvent。

事件对象封装了事件源与监听器彼此通信的事件信息。在必要的时候，可以对传递给监听器对象的事件对象进行分析。在按钮例子中，是借助getSource和getActionCommand方法实现对象分析的。

对于有些AWT事件类来说，Java程序员并不会实际地使用它们。例如，AWT将会把PaintEvent对象插入事件队列中，但这些对象并没有传递给监听器。Java程序员并不监听绘图事件，如果希望控制重新绘图操作，就需要覆盖paintComponent方法。另外，AWT还可以生成许多只对系统程序员有用的事件，用于提供表义语言的输入系统以及自动检测机器人等等。在此，将不讨论这些事件类型。

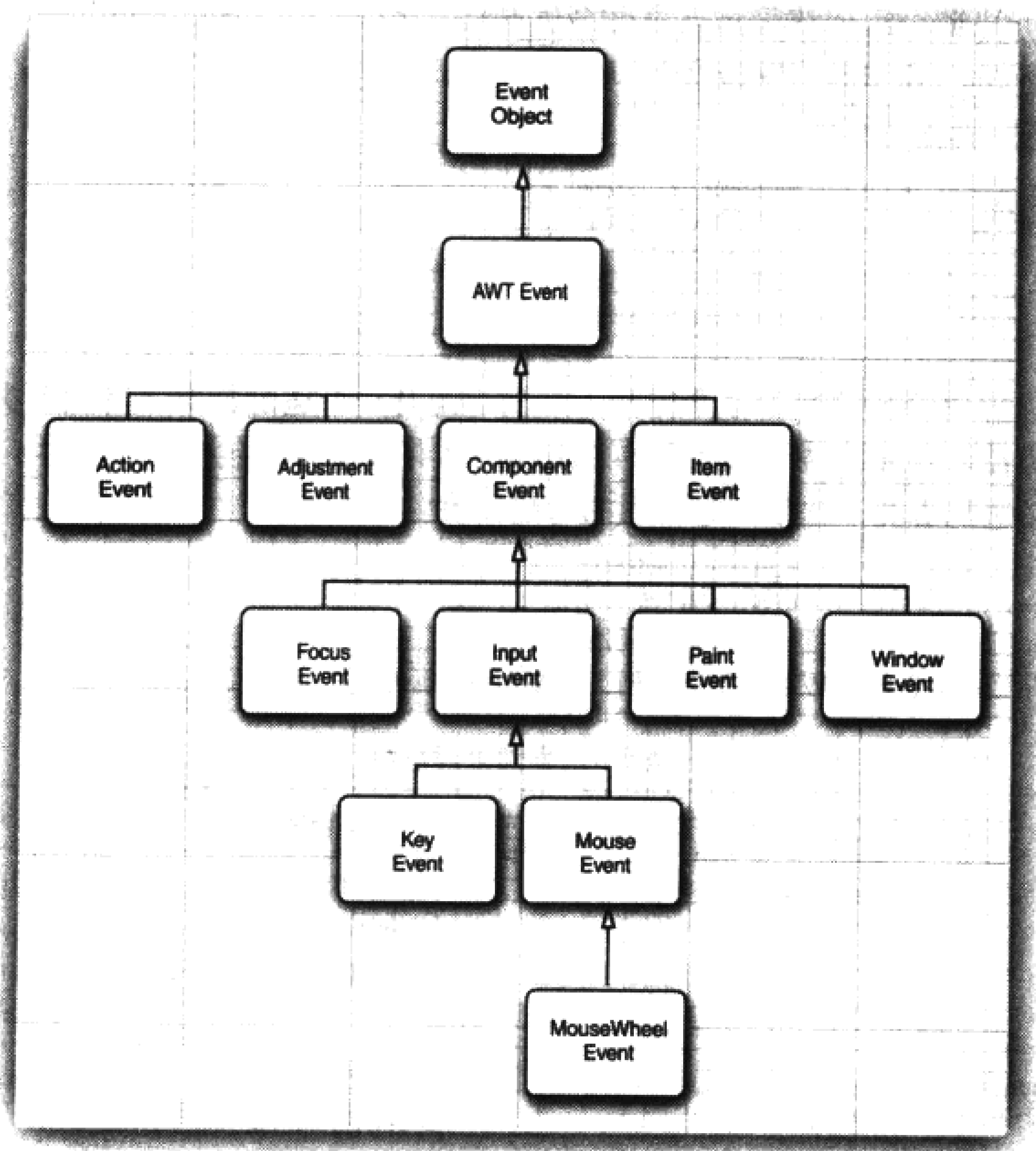


图8-8 AWT事件类的继承关系图

AWT的语义事件和低级事件

AWT将事件分为低级（low-level）事件和语义（semantic）事件。语义事件是表示用户动作的事件，例如，点击按钮；因此，ActionEvent是一种语义事件。低级事件是形成那些事件的事件。在点击按钮时，包含了按下鼠标、连续移动鼠标、抬起鼠标（只有鼠标在按钮区中抬起才引发）事件。或者在用户利用TAB键选择按钮，并利用空格键激活它时，发生的敲击键盘事件。同样，调节滚动条是一种语义事件，但拖动鼠标是低级事件。

下面是java.awt.event包中最常用的语义事件类：

- ActionEvent（对应按钮点击、菜单选择、选择列表项或在文本框中ENTER）；
- AdjustmentEvent（用户调节滚动条）；
- ItemEvent（用户从复选框或列表框中选择一项）。

常用的5个低级事件类是：

- KeyEvent（一个键被按下或释放）；
- MouseEvent（鼠标键被按下、释放、移动或拖动）；
- MouseWheelEvent（鼠标滚轮被转动）；

- FocusEvent（某个组件获得焦点或失去焦点）；
- WindowEvent（窗口状态被改变）。

下列接口将监听这些事件。

```
ActionListener            MouseMotionListener
AdjustmentListener        MouseWheelListener
FocusListener             WindowListener
ItemListener              WindowFocusListener
KeyListener               WindowStateListener
MouseListener
```

有几个AWT监听器接口包含多个方法，它们都配有一个适配器类，在这个类中实现了相应接口中的所有方法，但每个方法没有做任何事情。（有些接口只包含一个方法，因此，就没有必要为它们定义适配器类了）下面是常用的适配器类：

```
FocusAdapter              MouseMotionAdapter
KeyAdapter                WindowAdapter
MouseAdapter
```

表8-4显示了最重要的AWT监听器接口、事件和事件源。

表8-4 事件处理总结

接 口	方 法	参数/访问方法	事 件 源
ActionListener	actionPerformed	ActionEvent • getActionCommand • getModifiers	AbstractButton JComboBox JTextField Timer
AdjustmentListener	adjustmentValueChanged	AdjustmentEvent • getAdjustable • getAdjustmentType • getValue	JScrollbar
ItemListener	itemStateChanged	ItemEvent • getItem • getItemSelectable • getStateChange	AbstractButton JComboBox
FocusListener	focusGained focusLost	FocusEvent • isTemporary	Component
KeyListener	keyPressed keyReleased keyTyped	KeyEvent • getKeyChar • getKeyCode • getKeyModifiersText • getKeyText • isActionKey	Component
MouseListener	mousePressed mouseReleased mouseEntered	MouseEvent • getClickCount • getX	Component

（续）

接口	方法	参数/访问方法	事件源
	mouseExited mouseClicked	• getY • getPoint • translatePoint	
MouseMotionListener	mouseDragged mouseMoved	MouseEvent	Component
MouseWheelListener	mouseWheelMoved	MouseWheelEvent • getWheelRotation • getScrollAmount	Component
WindowListener	windowClosing windowOpened windowIconified windowDeiconified windowClosed windowActivated windowDeactivated	WindowEvent • getWindow	Window
WindowFocusListener	windowGainedFocus windowLostFocus	WindowEvent • getOppositeWindow	Window
WindowStateListener	windowStateChanged	WindowEvent • getOldState • getNewState	Window

javax.swing.event包中包含了许多专门用于Swing组件的附加事件，下一章中将介绍其中的一部分。

AWT事件处理的讨论到此结束。在下一章中，读者将学习Swing提供的更多的常用组件，并详细地介绍它们所产生的事件。

第9章 Swing用户界面组件

- ▲ Swing与模型-视图-控制器设计模式
- ▲ 布局管理概述
- ▲ 文本输入
- ▲ 选择组件
- ▲ 菜单
- ▲ 复杂的布局管理
- ▲ 对话框

上一章主要介绍了如何使用Java中的事件模式。通过学习读者已经初步知道了构造图形用户界面的基本方法。本章将介绍构造功能更加齐全的图形用户界面所需要的一些重要工具。

下面，首先介绍Swing的基本体系结构。要想弄清如何有效地使用一些更高级的组件，必须了解底层的东西。然后，再讲述Swing中各种常用的用户界面组件，如文本框、单选按钮以及菜单等等。接下来，介绍在不考虑特定的用户界面观感时，如何使用Java中的布局管理器排列在窗口中的这些组件。最后，介绍如何在Swing中实现对话框。

本章囊括了基本的Swing组件，如文本组件、按钮和滑块等，这些都是基本的用户界面组件，使用十分频繁。Swing中的高级组件将在卷II中讨论。

9.1 Swing和模型-视图-控制器设计模式

前面说过，本章将从Swing组件的体系结构开始。首先，我们讨论一下设计模式的概念，然后再看一下Swing框架中最具影响力的“模型-视图-控制器”模式。

9.1.1 设计模式

在解决一个问题时，不需要从头做起，而是借鉴过去的经验，或者向做过相关工作的专家请教。设计模式就是一种方法，这种方法以一种结构化的方式展示专家们的心血。

近几年来，软件工程师们开始对这些模式进行汇总分类。这个领域的先驱者的灵感来源于建筑师Christopher Alexander的设计模式。他在《The Timeless Way of Building》（1979年，牛津大学出版）一书中，为公共和私人居住空间的建筑设计模式进行了分类。下面是一个典型的例子：

窗户位置

每个人都喜欢靠窗户的座位，可以画上凸出去的窗户、低窗台的大窗户以及放在这里的舒适椅子。如果一个房间中没有这样一个地方，很少有人会感到舒服和安逸。

如果房间中没有像这样“位置”的一个窗户，房间里的人就有可能要做出下列抉择：（1）舒适地坐下；（2）要充足的阳光。

显然，舒适的地方是房间中最想坐的地方，但它们远离窗户，这种冲突不可避免。

因此，对于白天长时间逗留的房间，至少要将一个窗户开在“窗户位置”处。

在Alexander的模式分类和软件模式的分类中，每种模式都遵循一种特定的格式。这些模式首先描述背景，即引发设计问题的情形；接着解释问题，通常这里会有几个冲突的因素；最终，权衡这些冲突，给出问题的解决方案。

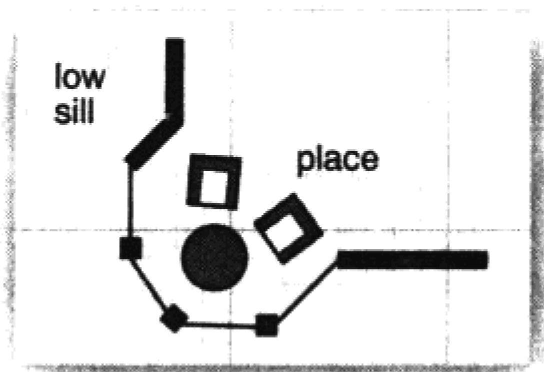


图9-1 窗户位置

在“窗户位置”模式中，背景是在白天逗留时间较长的房间。冲突因素就是既想舒适地坐下，又想拥有充足的光线。解决方案是找到一个“窗户位置”。

在“模型-视图-控制器”模式中，背景是显示信息和接收用户输入的用户界面系统。有关“模型-视图-控制器”模式将在接下来的章节中讲述。这里有几个冲突因素。对于同一数据来说，可能需要同时更新多个可视化表示。例如，为了适应各种观感标准，可能需要改变可视化表示形式；又例如，为了支持语音命令，可能需要改变交互机制。解决方案是将这些功能分布到三个独立的交互组件：模型、视图和控制器。

模型-视图-控制器模式并不是AWT和Swing设计中使用的惟一模式。下列是应用的另外几种模式：

- 容器和组件是“组合（composite）”模式
- 带滚动条的面板是“装饰（decorator）”模式
- 布局管理器是“策略（strategy）”模式

设计模式的另外一个最重要的特点是它们已经成为文化的一部分。只要谈论起模型-视图-控制器或“装饰”模式，遍及世界各地的程序员就会明白。因此，模式已经成为探讨设计方案的一种有效方法。

读者可以在Erich Gamma等编著的《Design Patterns——Elements of Reusable Object-Oriented Software》（Addison-Wesley出版社，1995年出版[⊖]）一书中找到大量的实用软件模式的规范描述，这是一本研究模式运动的书籍。这里再强烈地推荐一本由Frank Buschmann等编著的《A System of Patterns》，John Wiley & Sons出版社于1996出版。这是一本不错的书籍，相对前一本，这本书更容易读懂。

9.1.2 模型-视图-控制器模式

让我们稍稍停顿一会儿，回想一下构成用户界面组件的各个组成部分，例如，按钮、复选框、文本框或者复杂的树形组件等。每个组件都有三个要素：

- 内容，如：按钮的状态（是否按下），或者文本框的文本。
- 外观（颜色，大小等等）。
- 行为（对事件的反应）。

这三个要素之间的关系是相当复杂的，即使对于最简单的组件（如：按钮）来说也是如此。很明显，按钮的外观显示取决于它的观感。Metal按钮的外观与Windows按钮或者Motif按钮的外观就不一样。另外，外观显示还要取决于按钮的状态：当按钮被按下时，按钮需要被重新绘制成

⊖ 本书中文版《设计模式-可复用面向对象软件的基础》已由机械工业出版社出版。——编辑注

另一种不同的外观。而状态取决于按钮接收到的事件。当用户在按钮上点击时，按钮就被按下。

当然，在程序中使用按钮时，只需要简单地把它看成是一个*按钮*，而不需要考虑它的内部工作和特性。毕竟，这些是实现按钮的程序员的工作。无论怎样，实现按钮的程序员就要对这些按钮考虑得细致一些了。毕竟，无论观感如何，他们必须实现这些按钮和其他用户界面组件，以便让这些组件正常地工作。

为了实现这样的需求，Swing设计者采用了一种很有名的*设计模式*（design pattern）：*模型－视图－控制器*（model-view-controller）模式。这种设计模式同其他许多设计模式一样，都遵循第5章介绍过的面向对象设计中的一个基本原则：限制一个对象拥有的功能数量。不要用一个按钮类完成所有的事情，而是应该让一个对象负责组件的观感，另一个对象负责存储内容。模型-视图-控制器（MVC）模式告诉我们如何实现这种设计，实现三个独立的类：

- *模型*（model）：存储内容。
- *视图*（view）：显示内容。
- *控制器*（controller）：处理用户输入。

这个模式明确地规定了三个对象如何进行交互。模型存储内容，它*没有用户界面*。按钮的内容非常简单，只有几个用来表示当前按钮是否按下，是否处于活动状态的标志等。文本框内容稍稍复杂一些，它是保存当前文本的字符串对象。这与视图显示的内容并不一致——如果内容的长度大于文本框的显示长度，用户就只能看到文本框可以显示的那一部分，如图9-2所示。

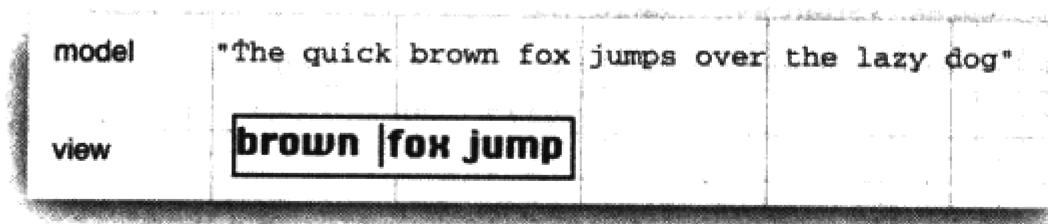


图9-2 文本框的模型和视图

模型必须实现改变内容和查找内容的方法。例如，一个文本模型中的方法有：在当前文本中添加或者删除字符以及把当前文本作为一个字符串返回等。记住：模型是完全不可见的。显示存储在模型中的数据是视图的工作。

注释：“模式”这个术语可能不太贴切，因为人们通常把模式视为一个抽象概念的具体表示。汽车和飞机的设计者构造模式来模拟真实的汽车和飞机。但这种类比可能会使你对模型-视图-控制器模式产生错误的理解。在设计模式中，模型存储完整的内容，视图给出了内容的可视化显示（完整或者不完整）。一个更恰当的比喻应当是模特为画家摆好姿势。此时，就要看画家如何看待模特，并由此来作一张画了。那张画是一幅规矩的肖像画，或是一幅印象派作品，还是一幅立体派作品（以古怪的曲线来描绘四肢）完全取决于画家。

模型－视图－控制器模式的一个优点是一个模型可以有多个视图，其中每个视图可以显示全部内容的不同部分或不同形式。例如，一个HTML编辑器常常为同一内容在同一时刻提供两个视图：一个WYSIWYG（所见即所得）视图和一个“原始标记”视图（见图9-3）。当通过某一个视图的控制器对模型进行更新时，模式会把这种改变通知给两个视图。视图得到通知以后

就会自动地刷新。当然，对于一个简单的用户界面组件来说，如按钮，不需要为同一模型提供多个视图。

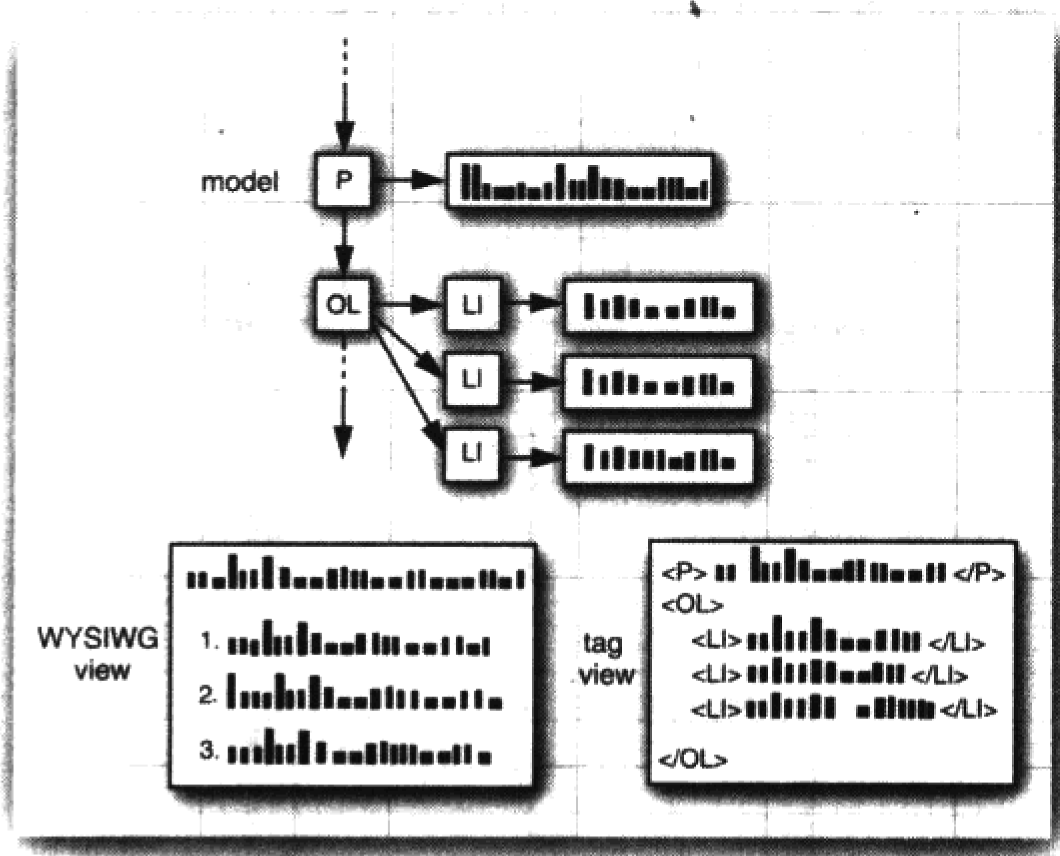


图9-3 同一模型的两个不同视图

控制器负责处理用户输入事件，如点击鼠标和敲击键盘。然后决定是否把这些事件转化成对模型或视图的改变。例如，如果用户在一个文本框中按下了一个字符键，控制器调用模型中的"插入字符"命令，然后模型告诉视图进行更新，而视图永远不会知道文本为什么改变了。但是如果用户按下了一个光标键，那么控制器会通知视图进行卷屏。卷动视图对实际文本不会有任何影响，因此模型永远不会知道这个事件的发生。

图9-4给出了模型、视图和控制器对象之间的交互。

在程序员使用Swing组件时，通常不需要考虑模型-视图-控制器体系结构。每个用户界面元素都有一个包装器类（如JButton或JTextField）来保存模型和视图。当需要查询内容（如文本域中的文本）时，包装器类会向模型询问并且返回所要的结果。当想改变视图时（例如，在一个文本域中移动光标位置），包装器类会把此请求转发给视图。然而，有时候包装器转发命令并不得力。在这种情况下，就必须直接地与模型打交道（不必直接操作视图——这是观感代码的任务）。

除了"本职工作"外，模型-视图-控制器模式吸引Swing设计者的主要原因是这种模式允许实现可插观感。每个按钮或者文本域的模型是独立于观感的。当然可视化表示完全依赖于特殊观感的用户界面设计，且控制器可以改变它。例如，在一个语音控制设备中，控制器需要处理的各种事件与使用键盘和鼠标的标准计算机完全不同。通过把底层模型与用户界面分离开，Swing设计者就能够重用模型的代码，甚至在程序运行时对观感进行切换。

当然，模式只能作为一种指导性的建议而并没有严格的戒律。没有一种模式能够适用于所

有情况。例如，使用“窗户位置”模式（设计模式中并非主要成分）来安排小卧室就不太合适。同样地，Swing设计者发现对于可插观感实现来说，使用模型-视图-控制器模式并非都是完美的。模型容易分离开，每个用户界面组件都有一个模型类。但是，视图和控制器的职责分工有时就不很明显，这样将会导致产生很多不同的类。当然，作为这些类的使用者来说，不必为这些细节费心。前面已经说过，这些类的使用者根本无需为模型操心，仅使用组件包装器类即可。

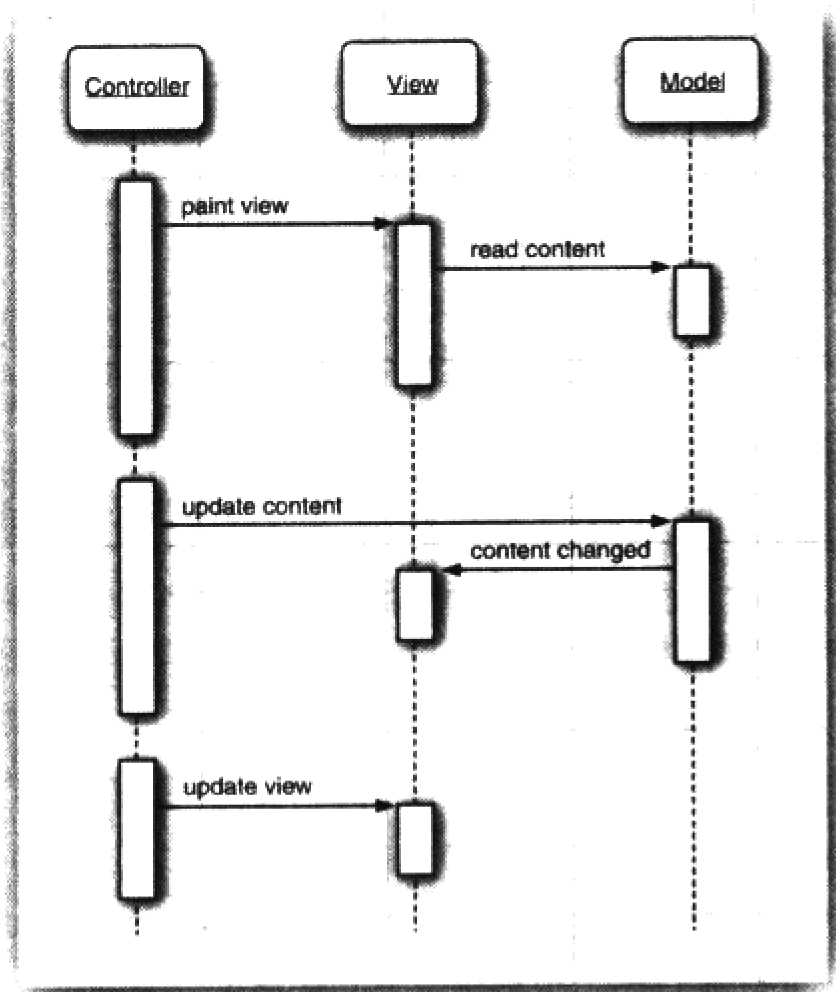


图9-4 给出了模型、视图、控制器对象之间的交互。

9.1.3 Swing按钮的模型–视图–控制器分析

前一章已经介绍了如何使用按钮，当时没有考虑模型、视图和控制器。按钮是最简单的用户界面元素，所以我们从按钮开始学习模型–视图–控制器模式会感觉容易些。对于更复杂的Swing组件来说，所遇到的类和接口都是类似的。

对于大多数组件来说，模型类将实现一个名字以Model结尾的接口，例如，按钮就实现了ButtonModel接口。实现了此接口的类可以定义各种按钮的状态。实际上，按钮并不复杂，在Swing库中有一个名为DefaultButtonModel的类就实现了这个接口。

读者可以通过查看ButtonModel接口中的特征来了解按钮模型所维护的数据类别。表9-1列出了这些特征。

每个JButton对象都存储了一个按钮模型对象，可以用下列方式得到它的引用。

```
JButton button = new JButton("Blue");
ButtonModel model = button.getModel();
```

示 汽车和飞机的设计者构造模式来模拟真实的汽车和飞机。但这种类比可能会使你对模

表9-1 ButtonModel接口的属性

属性名	值
ActionCommand	与按钮关联的动作命令字符串
Mnemonic	按钮的快捷键
Armed	如果按钮被按下且鼠标仍在按钮上则为true
Enabled	如果按钮是可选择的则为true
Pressed	如果按钮被按下且鼠标按键没有释放则为true
Rollover	如果鼠标在按钮之上则为true
Selected	如果按钮已经被选择（用于复选框和单选按钮）则为true

实际上，不必关注按钮状态的零散信息，只有绘制它的视图才对此感兴趣。诸如按钮是否可用这样的重要信息完全可以通过JButton类得到（当然，JButton类也通过向它的模型询问来获得这些信息）。

下面查看一下ButtonModel接口中不包含的信息。模型不存储按钮标签或者图标。对于一个按钮来说，仅凭模型无法知道它的外观（实际上，在有关单选按钮组的一节中将会看到，这种纯粹的设计会给程序员带来一些麻烦）。

需要注意的是，同样的模型（即DefaultButtonModel）可用于下压按钮、单选按钮、复选框、甚至是菜单项。当然，这些按钮都有各自不同的视图和控制器。当使用Metal观感时，JButton类用BasicButtonUI类作为其视图；用ButtonUIListener类作为其控制器。通常，每个Swing组件都有一个相关的后缀为UI的视图对象，但并不是所有的Swing组件都有专门的控制器对象。

在阅读JButton底层工作的简介之后可能会想到：JButton究竟是什么？事实上，它仅仅是一个继承了JComponent的包装器类，JComponent包含了一个DefaultButtonModel对象，一些视图数据（例如按钮标签和图标）和一个负责按钮视图的BasicButtonUI对象。

9.2 布局管理器概述

在讨论每个Swing组件（例如：文本域和单选按钮）之前，首先介绍一下如何把这些组件排列在一个框架内。与Visual Basic不同，由于在JDK中没有表单设计器，所以需要通过编写代码来定制（布局）用户界面组件所在的位置。

当然，如果有支持Java的开发环境，就可能有某种布局工具来部分自动地或全部自动地完成这些布局任务。然而，弄清底层的实现方式是非常重要的，因为即使最好的工具有时也需要手工编码。

回顾上一章的程序，我们设计了几个按钮，点击这些按钮可以改变框架的背景颜色。所图9-5所示。

这几个按钮被放置在一个JPanel对象中，且用流布局管理器（flow layout manager）管理，这是面板的默认布局管理器。图9-6展示了向面板中添加多个按钮后的效果。正如读者所看到的，当一行的空间不够时，会将显示在新的一行上。

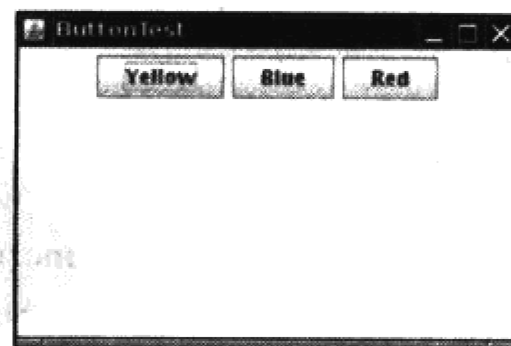


图9-5 包含三个按钮的面板

另外，按钮总是位于面板的中央，即使用户对框架进行缩放也是如此。如图9-7所示。

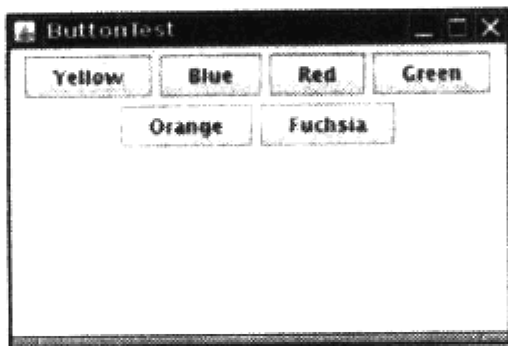


图9-6 用流布局管理六个按钮的面板

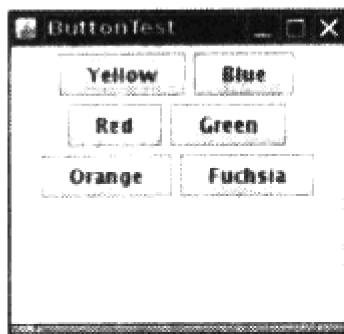


图9-7 改变面板尺寸后自动重新安排按钮

通常，组件放置在容器中，布局管理器决定容器中的组件具体放置的位置和大小。

按钮、文本域和其他的用户界面元素都继承于Component类，组件可以放置在面板这样的容器中。由于Container类继承于Component类，所以容器也可以放置在另一个容器中。图9-8给出了Component的类层次结构。

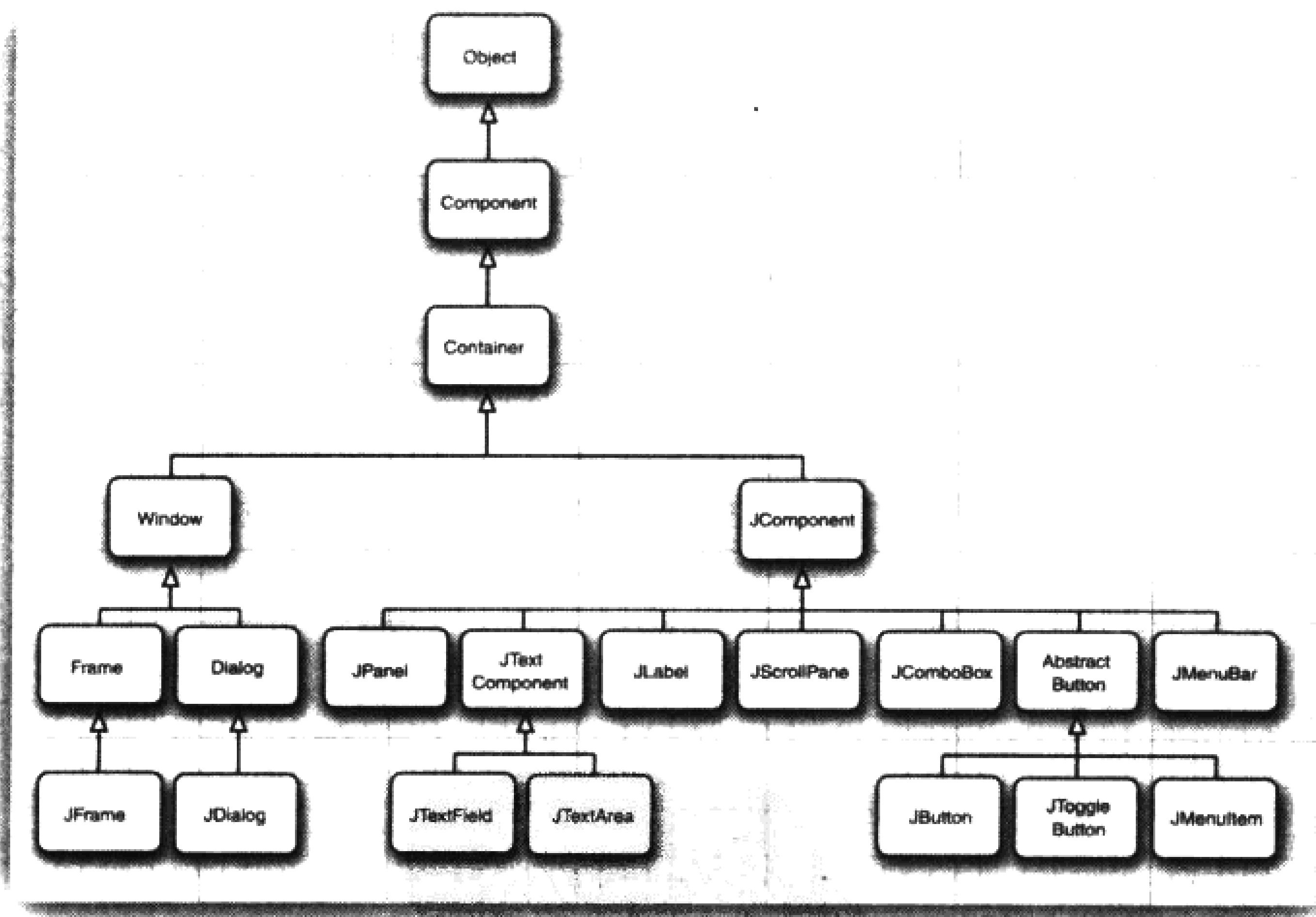


图9-8 Component的类层次结构

注释：可惜的是，继承层次有两点显得有点混乱。首先，像JFrame这样的顶层窗口是Container的子类，所以也是Component的子类，但却不能放在其他容器内。另外，JComponent是Container的子类，但不直接继承Component，因此，可以将其他组件添置到JButton中。(但无论如何，这些组件无法显示出来)。

每个容器都有一个默认的布局管理器，但可以重新进行设置。例如，使用下列语句：

```
panel.setLayout(new GridLayout(4, 4));
```

这个面板将用GridLayout类布局组件。可以往容器中添加组件。容器的add方法将把组件和放置的方位传递给布局管理器。

API java.awt.Container 1.0

- `Void SetLayout (LayoutManager m)`
  为容器设置布局管理器
- `Component add(Component c)`
- `Component add(Component c, Object constraints) 1.1`
  将组件添加到容器中，并返回组件的引用。

参数：	
c	要添加的组件
constraints	布局管理器理解的标识符

API java.awt.FlowLayout 1.0

- `FlowLayout ()`
- `FlowLayout (int align)`
- `FlowLayout (int align, int hgap, int vgap)`
  构造一个新的FlowLayout对象。

参数：	
align	LEFT、CENTER或者RIGHT
hgap	以像素为单位的水平间距（如果为负值，则强行重叠）
vgap	以像素为单位的垂直间距（如果为负值，则强行重叠）

9.2.1 边框布局

边框布局管理器（border layout manager）是每个JFrame的内容窗格的默认布局管理器。流布局管理器完全控制每个组件的放置位置，边框布局管理器则不然，它允许为每个组件选择一个放置位置。可以选择把组件放在内容窗格的中部、北部、南部、东部或者西部。如图9-9所示。

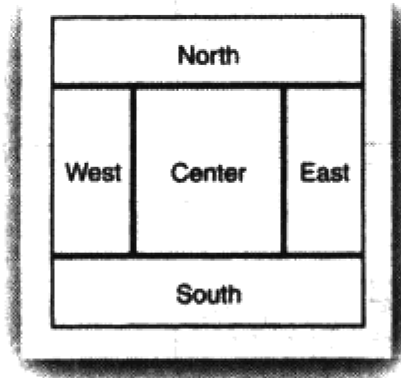


图9-9 边框布局

例如：

```
frame.add(component, BorderLayout.SOUTH);
```

先放置边缘组件，剩余的可用空间由中间组件占据。当容器被缩放时，边缘组件的厚度不会改变，而中部组件的大小会发生变化。在添加组件时可以指定BorderLayout类中的CENTER、NORTH、SOUTH、EAST和WEST常量。并非需要占用所有的位置，如果没有提供任何值，系统默认为CENTER。

> **注释**：BorderLayout常量定义为字符串。例如：BorderLayout.SOUTH定义为字符串"SOUTH"。很多程序员喜欢直接使用字符串，因为这些字符串比较简短，例如，frame.add (component, "SOUTH")。然而，如果字符串拼写有误，编译器不会捕获错误。

与流布局不同，边框布局会扩展所有组件的尺寸以便填满可用空间（流布局将维持每个组件的最佳尺寸）。当将一个按钮添加到容器中时会出现问题：

```
frame.add(yellowButton, BorderLayout.SOUTH); // don't
```

图9-10给出了执行上述语句的显示效果。按钮扩展至填满框架的整个南部区域。而且，如果再将另外一个按钮添加到南部区域，就会取代第一个按钮。

解决这个问题的常见方法是使用另外一个面板（panel）。例如，如图9-11所示。屏幕底部的三个按钮全部包含在一个面板中。这个面板被放置在内容窗格的南部。

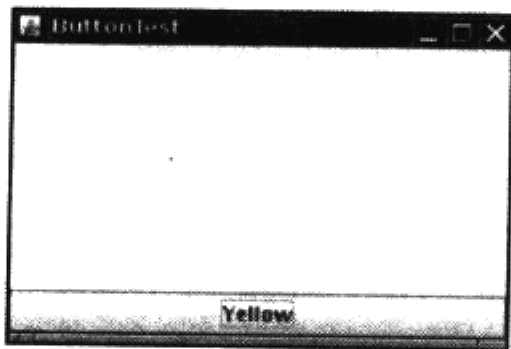


图9-10 边框布局管理一个按钮

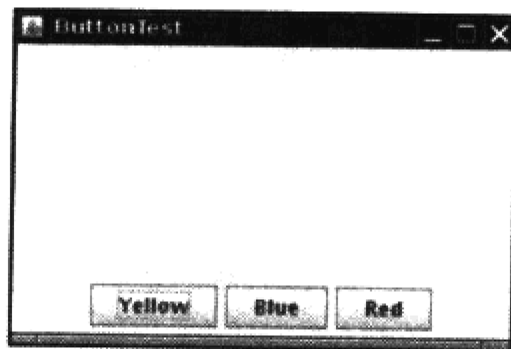


图9-11 面板放置在框架的南部区域

要想得到这种效果，首先需要创建一个新的JPanel对象，然后逐一将按钮添加到面板中。面板的默认布局管理器是FlowLayout，这恰好符合我们的需求。随后使用在前面已经看到的add方法将每个按钮添加到面板中。每个按钮的放置位置和尺寸完全处于FlowLayout布局管理器的控制之下。这意味着这些按钮将置于面板的中央，并且不会扩展至填满整个面板区域。最后，将这个面板添加到框架的内容窗格中。

```
JPanel panel = new JPanel();
panel.add(yellowButton);
panel.add(blueButton);
panel.add(redButton);
frame.add(panel, BorderLayout.SOUTH);
```

边框布局管理器将会扩展面板大小，直至填满整个南部区域。

API java.awt.BorderLayout 1.0

- `BorderLayout()`
- `BorderLayout( int hgap, int vgap)`

  构造一个新的BorderLayout对象。

  参数：hgap　以像素为单位的水平间距（如果为负值，则强行重叠）

  　　　vgap　以像素为单位的垂直间距（如果为负值，则强行重叠）

9.2.2 网格布局

网格布局像电子数据表一样，按行列排列所有的组件。不过，它的每个单元大小都是一样的。图9-12显示的计算器程序就使用了网格布局来排列计算器按钮。当缩放窗口时，计算器按钮将随之变大或变小，但所有的按钮尺寸始终保持一致。

在网格布局对象的构造器中，需要指定行数和列数：

```
panel.setLayout(new GridLayout(5, 4));
```

添加组件，从第一行的第一列开始，然后是第一行的第二列，以此类推。

```
panel.add(new JButton("1"));
panel.add(new JButton("2"));
```

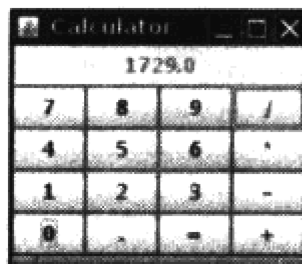


图9-12 计算器

例9-1是计算器程序的源代码。这是一个常规的计算器，而不像Java指南中所提到的“逆波兰”那样古怪。在这个程序中，在将组件添加到框架之后，调用了pack方法。这个方法用于将所有组件以最佳的高度和宽度显示在框架中。

当然，极少有像计算器这样整齐的布局。实际上，在组织窗口的布局时小网格（通常只有一行或者一列）比较有用。例如，如果想放置一行尺寸都一样的按钮，就可以将这些按钮放置在一个面板里，这个面板使用只有一行的网格布局进行管理。

例9-1 Calculator.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * @version 1.33 2007-06-12
 * @author Cay Horstmann
 */
public class Calculator
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               CalculatorFrame frame = new CalculatorFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * A frame with a calculator panel.
 */
class CalculatorFrame extends JFrame
{
   public CalculatorFrame()
   {
      setTitle("Calculator");
      CalculatorPanel panel = new CalculatorPanel();
      add(panel);
      pack();
   }
```

```
}

/**
 * A panel with calculator buttons and a result display.
 */
class CalculatorPanel extends JPanel
{
   public CalculatorPanel()
   {
      setLayout(new BorderLayout());

      result = 0;
      lastCommand = "=";
      start = true;

      // add the display

      display = new JButton("0");
      display.setEnabled(false);
      add(display, BorderLayout.NORTH);

      ActionListener insert = new InsertAction();
      ActionListener command = new CommandAction();

      // add the buttons in a 4 x 4 grid

      panel = new JPanel();
      panel.setLayout(new GridLayout(4, 4));

      addButton("7", insert);
      addButton("8", insert);
      addButton("9", insert);
      addButton("/", command);

      addButton("4", insert);
      addButton("5", insert);
      addButton("6", insert);
      addButton("*", command);

      addButton("1", insert);
      addButton("2", insert);
      addButton("3", insert);
      addButton("-", command);

      addButton("0", insert);
      addButton(".", insert);
      addButton("=", command);
      addButton("+", command);

      add(panel, BorderLayout.CENTER);
   }

   /**
    * Adds a button to the center panel.
    * @param label the button label
    * @param listener the button listener
    */
```

```
   private void addButton(String label, ActionListener listener)
   {
      JButton button = new JButton(label);
      button.addActionListener(listener);
      panel.add(button);
   }

   /**
    * This action inserts the button action string to the end of the display text.
    */
   private class InsertAction implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         String input = event.getActionCommand();
         if (start)
         {
            display.setText("");
            start = false;
         }
         display.setText(display.getText() + input);
      }
   }

   /**
    * This action executes the command that the button action string denotes.
    */
   private class CommandAction implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         String command = event.getActionCommand();

         if (start)
         {
            if (command.equals("-"))
            {
               display.setText(command);
               start = false;
            }
            else lastCommand = command;
         }
         else
         {
            calculate(Double.parseDouble(display.getText()));
            lastCommand = command;
            start = true;
         }
      }
   }

   /**
    * Carries out the pending calculation.
    * @param x the value to be accumulated with the prior result.
    */
   public void calculate(double x)
   {
```

```
        if (lastCommand.equals("+")) result += x;
        else if (lastCommand.equals("-")) result -= x;
        else if (lastCommand.equals("*")) result *= x;
        else if (lastCommand.equals("/")) result /= x;
        else if (lastCommand.equals("=")) result = x;
        display.setText("" + result);
    }

    private JButton display;
    private JPanel panel;
    private double result;
    private String lastCommand;
    private boolean start;
}
```

API java.awt.GridLayout 1.0

- `GridLayout(int rows, int cols)`
- `GridLayout(int rows, int columns, int hgap, int vgap)`

  构造一个新的GridLayout对象。rows或者columns可以为零，但不能同时为零，指定的每行或每列的组件数量可以任意的。

  参数：rows　网格的行数

  columns　网格的列数

  hgap　以像素为单位的水平间距（如果为负值，则强行重叠）

  vgap　以像素为单位的垂直间距（如果为负值，则强行重叠）

API java.awt.Window 1.0

- `void pack()`

  缩放窗口时，将组件调整至最佳尺寸。

9.3 文本输入

现在终于可以开始介绍Swing用户界面组件了。首先，介绍具有用户输入和编辑文本功能的组件。文本域（JTextField）和文本区（JTextArea）组件用于获取文本输入。文本域只能接收单行文本的输入，而文本区能够接收多行文本的输入。JPassword也只能接收单行文本的输入，但不会将输入的内容显示出来。

这三个类都继承于JTextComponent类。由于JTextComponent是一个抽象类，所以不能够构造这个类的对象。另外，在Java中常会看到这种情况。在查看API文档时，发现自己正在寻找的方法实际上来自于父类JTextComponent，而不是来自派生类自身。例如，在一个文本域和文本区内获取（get）、设置（set）文本的方法实际上都是JTextComponent类中的方法。

API javax.swing.text.JTextComponent 1.2

- `String getText()`

- void setText(String text)

  获取或设置文本组件中的文本。
- boolean isEditable()
- void setEditable(boolean b)

  获取或设置editable特性，这个特性决定了用户是否可以编辑文本组件中的内容。

9.3.1 文本域

把文本域添加到窗口的常用办法是将它添加到面板或者其他容器中，这与添加按钮完全一样：

```
JPanel panel = new JPanel();
JTextField textField = new JTextField("Default input", 20);
panel.add(textField);
```

这段代码将添加一个文本域，同时通过传递字符串“Default input”进行初始化。构造器的第二个参数设置了文本域的宽度。在这个示例中，宽度值为20“列”。但是，这里所说的列不是一个精确的测量单位。一列就是在当前使用的字体下一个字符的宽度。如果希望文本域最多能够输入n个字符，就应该把宽度设置为n列。在实际中，这样做效果并不理想，应该将最大输入长度再多设1~2个字符。列数只是给AWT设定首选（preferred）大小的一个提示。如果布局管理器需要缩放这个文本域，它会调整文本域的大小。在JTextField的构造器中设定的宽度并不是用户能输入的字符个数的上限。用户可以输入一个更长的字符串，但是当文本长度超过文本域长度时输入就会滚动。用户通常不喜欢滚动文本域，因此应该尽量把文本域设置的宽一些。如果需要在运行时重新设置列数，可以使用setColumns方法。

提示：使用setColumns方法改变了一个文本域的大小之后，需要调用包含这个文本框的容器的revalidate方法。

```
textField.setColumns(10);
panel.revalidate();
```

revalidate方法会重新计算容器内所有组件的大小，并且对它们重新进行布局。调用revalidate方法以后，布局管理器会重新设置容器的大小，然后就可以看到改变尺寸后的文本域了。

revalidate方法是JComponent类中的方法。它并不是马上就改变组件大小，而是给这个组件加一个需要改变大小的标记。这样就避免了多个组件改变大小时带来的重复计算。但是，如果想重新计算一个JFrame中的所有组件，就必须调用validate方法——JFrame没有扩展JComponent。

通常情况下，希望用户在文本域中键入文本（或者编辑已经存在的文本）。文本域一般初始为空白。只要不为JTextField构造器提供字符串参数，就可以构造一个空白文本域：

```
JTextField textField = new JTextField(20);
```

可以在任何时候调用setText方法改变文本域中的内容。这个方法是从前面提到的JTextComponent中继承而来的。例如：

```
textField.setText("Hello!");
```

并且，在前面已经提到，可以调用getText方法来获取用户键入的文本。这个方法返回用户输入的文本。如果想要将getText方法返回的文本域中的内容的前后空格去掉，就应该调用trim方法：

```
String text = textField.getText().trim();
```

如果想要改变显示文本的字体，就调用setFont方法。

API javax.swing.JTextField 1.2

- `JTextField(int cols)`
  构造一个给定列数的空JTextField对象。
- `JTextField(String text, int cols)`
  构造一个给定列数、给定初始字符串的JTextField对象。
- `int getColumns()`
- `void setColumns(int cols)`
  获取或设置文本域使用的列数。

API javax.swing.JComponent 1.2

- `void revalidate()`
  重新计算组件的位置和大小。
- `void setFont(Font f)`
  设置组件的字体。

API java.awt.Component 1.0

- `void validate()`
  重新计算组件的位置和大小。如果组件是容器，容器中包含的所有组件的位置和大小也被重新计算。
- `Font getFont()`
  获取组件的字体。

9.3.2 标签和标签组件

标签是容纳文本的组件，它们没有任何的修饰（例如没有边缘），也不能响应用户输入。可以利用标签标识组件。例如：与按钮不同，文本域没有标识它们的标签。要想用标识符标识这种不带标签的组件，应该

1）用相应的文本构造一个JLabel组件。

2）将标签组件放置在距离需要标识的组件足够近的地方，以便用户可以知道标签所标识的组件。

JLabel的构造器允许指定初始文本和图标，也可以选择内容的排列方式。可以用Swing

Constants接口中的常量来指定排列方式。在这个接口中定义了几个很有用的常量，如LEFT、RIGHT、CENTER、NORTH、EAST等。JLabel是实现这个接口的一个Swing类。因此，可以指定右对齐标签：

```
JLabel label = new JLabel("User name: ", SwingConstants.RIGHT);
```

或者

```
JLabel label = new JLabel("User name: ", JLabel.RIGHT);
```

利用setText和setIcon方法可以在运行期间设置标签的文本和图标。

提示：从JDK1.3开始，可以在按钮、标签和菜单项上使用无格式文本或HTML文本。我们不推荐在按钮上使用HTML文本——这样会影响观感。但是HTML文本在标签中是非常有效的。只要简单地将标签字符串放置在<html>...</html>中即可：

```
label = new JLabel("<html><b>Required</b> entry:</html>");
```

警告——包含HTML标签的第一个组件需要延迟一段时间才能显示出来，这是因为需要加载相当复杂的HTML显示代码。

与其他组件一样，标签也可以放置在容器中。这就是说，可以利用前面介绍的技巧将标签放置在任何需要的地方。

API javax.swing.JLabel 1.2

- `JLabel(String text)`
- `JLabel(Icon icon)`
- `JLabel(String text, int align)`
- `JLabel(String text, Icon icon, int align)`

  构造一个标签。

  参数：text　　标签中的文本

  　　　icon　　标签中的图标

  　　　align　　一个SwingConstants的常量LEFT（默认），CENTER或者RIGHT

- `String getText()`
- `void setText(String text)`

  获取或设置标签的文本。

- `Icon getIcon()`
- `void setIcon(Icon icon)`

  获取或设置标签的图标。

9.3.3 密码域

密码域是一种特殊类型的文本域。为了避免有某种企图的人看到密码，用户输入的字符不显示出来。每个输入的字符都用*回显字符*（echo character）表示，典型的回显字符是星号（*）。Swing提供了JPasswordField类来实现这样的文本域。

密码域是另一个应用模型-视图-控制器体系模式的例子。密码域采用与常规的文本域相同的模型来存储数据，但是，它的视图却改为显示回显字符，而不是实际的字符。

API `javax.swing.JPasswordField 1.2`

- `JPasswordField(String text, int columns)`

  构造一个新的密码域对象。
- `void setEchoChar(char echo)`

  为密码域设置回显字符。注意：独特的观感可以选择自己的回显字符。0表示重新设置为默认的回显字符。
- `char[ ] getPassword()`

  返回密码域中的文本。为了安全起见，在使用之后应该覆写返回的数组内容（密码并不是以String的形式返回，这是因为字符串在被垃圾回收器回收之前会一直驻留在虚拟机中）。

9.3.4 文本区

有时，用户的输入超过一行。正像前面提到的，需要使用JTextArea组件来接收这样的输入。当在程序中放置一个文本区组件时，用户就可以输入多行文本，并用ENTER键换行。每行都以一个“\n”结尾。图9-13显示了一个工作的文本区。



图9-13 文本组件

在JTextArea组件的构造器中，可以指定文本区的行数和列数。例如：

```
textArea = new JTextArea(8, 40); // 8 lines of 40 columns each
```

与文本域一样。出于稳妥的考虑，参数columns应该设置的大一些。另外，用户并不受限于输入指定的行数和列数。当输入过长时，文本会滚动。还可以用setColumns方法改变列数，用setRows方法改变行数。这些数值只是首选大小——布局管理器可能会对文本区进行缩放。

如果文本区的文本超出显示的范围，那么剩下的文本就会被剪裁掉。可以通过开启换行特性来避免裁剪过长的行：

```
textArea.setLineWrap(true); // long lines are wrapped
```

换行只是视觉效果，文档中的文本没有改变，在文本中并没有插入“\n”字符。

9.3.5 滚动窗格

在Swing中，文本区没有滚动条。如果需要滚动条，可以将文本区插入到滚动窗格（scroll pane）中。

```
textArea = new JTextArea(8, 40);
JScrollPane scrollPane = new JScrollPane(textArea);
```

现在滚动窗格管理文本区的视图。如果文本超出了文本区可以显示的范围，滚动条就会自动地出现，并且在删除部分文本后，当文本能够显示在文本区范围内时，滚动条会再次自动地消失。滚动是由滚动窗格内部处理的，编写程序时无需处理滚动事件。

这是一种为任意组件添加滚动功能的通用机制，而不是文本区特有的。也就是说，要想为组件添加滚动条，只需将它们放入一个滚动窗格中即可。

例9-2展示了各种文本组件。这个程序只是简单地显示了一个文本域、一个密码域和一个带滚动条的文本区。文本域和密码域都使用了标签。点击“Insert”会将组件中的内容插入到文本区中。



注释：JTextArea组件只显示无格式的文本，没有字体或者格式设置。如果想要显示格式化文本（如HTML或者RTF），就需要使用JEditorPane和JTextPane类。在卷II将详细地讨论这几个类。

例9-2 TextComponentTest.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * @version 1.40 2007-04-27
 * @author Cay Horstmann
 */
public class TextComponentTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               TextComponentFrame frame = new TextComponentFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * A frame with sample text components.
 */
class TextComponentFrame extends JFrame
{
   public TextComponentFrame()
   {
      setTitle("TextComponentTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      final JTextField textField = new JTextField();
      final JPasswordField passwordField = new JPasswordField();

      JPanel northPanel = new JPanel();
      northPanel.setLayout(new GridLayout(2, 2));
      northPanel.add(new JLabel("User name: ", SwingConstants.RIGHT));
      northPanel.add(textField);
```

```
        northPanel.add(new JLabel("Password: ", SwingConstants.RIGHT));
        northPanel.add(passwordField);

        add(northPanel, BorderLayout.NORTH);

        final JTextArea textArea = new JTextArea(8, 40);
        JScrollPane scrollPane = new JScrollPane(textArea);

        add(scrollPane, BorderLayout.CENTER);

        // add button to append text into the text area

        JPanel southPanel = new JPanel();

        JButton insertButton = new JButton("Insert");
        southPanel.add(insertButton);
        insertButton.addActionListener(new ActionListener()
           {
              public void actionPerformed(ActionEvent event)
              {
                 textArea.append("User name: " + textField.getText() + " Password: "
                       + new String(passwordField.getPassword()) + "\n");
              }
           });

        add(southPanel, BorderLayout.SOUTH);

        // add a text area with scrollbars

     }

     public static final int DEFAULT_WIDTH = 300;
     public static final int DEFAULT_HEIGHT = 300;
  }
```



javax.swing.JTextArea 1.2

- `JTextArea()`
- `JTextArea(int rows, int cols)`
- `JTextArea(String text, int rows, int cols)`

  构造一个新的文本区对象。
- `void setColumns(int cols)`

  设置文本区应该使用的首选列数。
- `void setRows(int rows)`

  设置文本区应该使用的首选行数。
- `void append(String newText)`

  将给定的文本追加到文本区中已有文本的尾部。
- `void setLineWrap(boolean wrap)`

  打开或关闭换行。

• void setWrapStyleWord(boolean word)

如果word是true，超长的行会在字边框处换行。如果为false，超长的行被截断而不考虑字边框。

• void setTabSize(int c)

将制表符（tab stop）设置为c 列。注意，制表符不会被转化为空格，但可以让文本对齐到下一个制表符处。

API javax.swing.JScrollPane 1.2

• JScrollPane(Component c)

创建一个滚动窗格，用来显示指定组件的内容。当组件内容超过显示范围时，滚动条会自动地出现。

9.4 选择组件

前面已经讲述了如何获取用户输入的文本。然而，在很多情况下，可能更加愿意给用户几种选项，而不让用户在文本组件中输入数据。使用一组按钮或者选项列表让用户做出选择（这样也免去了检查错误的麻烦）。在本节中，将介绍如何编写程序来实现复选框、单选按钮、选项列表以及滑块。

9.4.1 复选框

如果想要接收的输入只是“是”或“非”，就可以使用复选框组件。复选框自动地带有标识标签。用户通过点击某个复选框来选择相应的选项，再点击则取消选取。当复选框获得焦点时，用户也可以通过按空格键来切换选择。

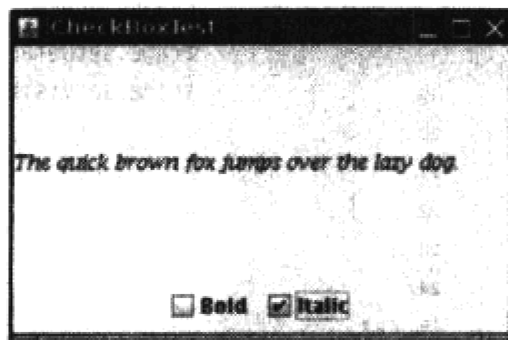


图9-14 复选框

图9-14所示的程序中有两个复选框，其中一个用于打开或关闭字体倾斜属性，而另一个用于控制加粗属性。注意，第二个复选框有焦点，这一点可以由它周围的矩形框看出。只要用户点击某个复选框，程序就会刷新屏幕以便应用新的字体属性。

复选框需要一个紧邻它的标签来说明其用途。在构造器中指定标签文本。

```
bold = new JCheckBox("Bold");
```

可以使用setSelected方法来选定或取消选定复选框。例如：

```
bold.setSelected(true);
```

isSelected方法将返回每个复选框的当前状态。如果没有选取则为false，否则为true。

当用户点击复选框时将触发一个动作事件。通常，可以为复选框设置一个动作监听器。在下面程序中，两个复选框使用了同一个动作监听器。

```
ActionListener listener = . . .
bold.addActionListener(listener);
italic.addActionListener(listener);
```

actionPerformed方法查询bold和italic两个复选框的状态，并且把面板中的字体设置为常规、

加粗、倾斜或者粗斜体。

```
public void actionPerformed(ActionEvent event)
{
   int mode = 0;
   if (bold.isSelected()) mode += Font.BOLD;
   if (italic.isSelected()) mode += Font.ITALIC;
   label.setFont(new Font("Serif", mode, FONTSIZE));
}
```

例9-3给出了复选框例子的全部代码。

例9-3 CheckBoxTest.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * @version 1.33 2007-06-12
 * @author Cay Horstmann
 */
public class CheckBoxTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               CheckBoxFrame frame = new CheckBoxFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * A frame with a sample text label and check boxes for selecting font attributes.
 */
class CheckBoxFrame extends JFrame
{
   public CheckBoxFrame()
   {
      setTitle("CheckBoxTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // add the sample text label

      label = new JLabel("The quick brown fox jumps over the lazy dog.");
      label.setFont(new Font("Serif", Font.PLAIN, FONTSIZE));
      add(label, BorderLayout.CENTER);

      // this listener sets the font attribute of
      // the label to the check box state

```

```
      ActionListener listener = new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               int mode = 0;
               if (bold.isSelected()) mode += Font.BOLD;
               if (italic.isSelected()) mode += Font.ITALIC;
               label.setFont(new Font("Serif", mode, FONTSIZE));
            }
         };

      // add the check boxes

      JPanel buttonPanel = new JPanel();

      bold = new JCheckBox("Bold");
      bold.addActionListener(listener);
      buttonPanel.add(bold);

      italic = new JCheckBox("Italic");
      italic.addActionListener(listener);
      buttonPanel.add(italic);

      add(buttonPanel, BorderLayout.SOUTH);
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;

   private JLabel label;
   private JCheckBox bold;
   private JCheckBox italic;

   private static final int FONTSIZE = 12;
}
```



javax.swing.JCheckBox 1.2

- `JCheckBox(String label)`
- `JCheckBox(String label, Icon icon)`

  构造一个复选框，初始没有被选择。
- `JCheckBox(String label, boolean state)`

  用给定的标签和初始化状态构造一个复选框。
- `boolean isSelected ()`
- `void setSelected(boolean state)`

  获取或设置复选框的选择状态。

9.4.2 单选按钮

在前一个例子中，对于两个复选框，用户既可以选择一个、两个，也可以两个都不选。在

很多情况下，我们需要用户只选择几个选项当中的一个。当用户选择另一项的时候，前一项就自动地取消选择。这样一组选框通常称为**单选按钮组**（Radio Button Group），这是因为这些按钮的工作很像收音机上的电台选择按钮。当按下一个按钮时，前一个按下的按钮就会自动弹起。图9-15给出了一个典型的例子。这里允许用户在多个选择中选择字体的大小，即小、中、大和超大，但是，每次用户只能选择一个。



图9-15 单选按钮组

在Swing中，实现单选按钮组非常简单。为单选按钮组构造一个ButtonGroup的对象。然后，再将JRadioButton类型的对象添加到按钮组中。按钮组负责在新按钮被按下时，取消前一个被按下的按钮的选择状态。

```
ButtonGroup group = new ButtonGroup();

JRadioButton smallButton = new JRadioButton("Small", false);
group.add(smallButton);

JRadioButton mediumButton = new JRadioButton("Medium", true);
group.add(mediumButton);
. . .
```

构造器的第二个参数为true表明这个按钮初始状态是被选择，其他按钮构造器的这个参数为false。注意，按钮组仅仅控制按钮的行为，如果想把这些按钮组织在一起布局，需要把它们添加到容器中，如JPanel。

如果再看一下图9-14和图9-15则会发现，单选按钮与复选框的外观是不一样的。复选框为正方形，并且如果被选择，这个正方形中会出现一个对钩的符号。单选按钮是圆形，选择以后圈内出现一个圆点。

单选按钮的事件通告机制与其他按钮一样。当用户点击一个单选按钮时，这个按钮将产生一个动作事件。在示例中，定义了一个动作监听器用来把字体大小设置为新值：

```
ActionListener listener = new
   ActionListener()
   {
   public void actionPerformed(ActionEvent event)
   {
      // size refers to the final parameter of the addRadioButton method
      label.setFont(new Font("Serif", Font.PLAIN, size));
   }
};
```

用这个监听器与复选框中的监听器做一个对比。每个单选按钮都对应一个不同的监听器对象。每个监听器都非常清楚所要做的事情——把字体尺寸设置为一个特定值。在复选框示例中，使用的是一种不同的方法，两个复选框共享一个动作监听器。这个监听器调用一个方法来检查两个复选框的当前状态。

对于单选按钮可以使用同一个方法吗？可以试一下使用一个监听器来计算尺寸，如：

```
if (smallButton.isSelected()) size = 8;
else if (mediumButton.isSelected()) size = 12;
. . .
```

然而，更愿意使用各自独立的动作监听器，因为这样可以将尺寸值与按钮紧密地绑定在一起。



注释：如果有一组单选按钮，并知道它们之中只选择了一个。要是能够不查询组内所有的按钮就可以很快地知道哪个按钮被选择的话就好了。由于ButtonGroup对象控制着所有的按钮，所以如果这个对象能够给出被选择的按钮的引用就方便多了。事实上，ButtonGroup类中有一个getSelection方法，但是这个方法并不返回被选择的单选按钮，而是返回附加在那个按钮上的模型ButtonModel的引用。对于我们来说，ButtonModel中的方法没有什么实际的应用价值。ButtonModel接口从ItemSelectable接口继承了一个getSelectedObject方法，但是这个方法没有用，它返回null。getActionCommand方法看起来似乎可用，这是因为一个单选按钮的“动作命令”是它的文本标签，但是它的模型的动作命令是null。只有在通过setActionCommand命令明确地为所有单选按钮设定动作命令后，才能够通过调用方法buttonGroup.getSelection().getActionCommand()获得当前选择的按钮的动作命令。

例9-4是一个用于选择字体大小的完整程序，它演示了单选按钮的工作过程。

例9-4 RadioButtonTest.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * @version 1.33 2007-06-12
 * @author Cay Horstmann
 */
public class RadioButtonTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               RadioButtonFrame frame = new RadioButtonFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * A frame with a sample text label and radio buttons for selecting font sizes.
 */
class RadioButtonFrame extends JFrame
```

```
{
   public RadioButtonFrame()
   {
      setTitle("RadioButtonTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // add the sample text label

      label = new JLabel("The quick brown fox jumps over the lazy dog.");
      label.setFont(new Font("Serif", Font.PLAIN, DEFAULT_SIZE));
      add(label, BorderLayout.CENTER);

      // add the radio buttons

      buttonPanel = new JPanel();
      group = new ButtonGroup();

      addRadioButton("Small", 8);
      addRadioButton("Medium", 12);
      addRadioButton("Large", 18);
      addRadioButton("Extra large", 36);

      add(buttonPanel, BorderLayout.SOUTH);
   }

   /**
    * Adds a radio button that sets the font size of the sample text.
    * @param name the string to appear on the button
    * @param size the font size that this button sets
    */
   public void addRadioButton(String name, final int size)
   {
      boolean selected = size == DEFAULT_SIZE;
      JRadioButton button = new JRadioButton(name, selected);
      group.add(button);
      buttonPanel.add(button);

      // this listener sets the label font size

      ActionListener listener = new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               // size refers to the final parameter of the addRadioButton
               // method
               label.setFont(new Font("Serif", Font.PLAIN, size));
            }
         };

      button.addActionListener(listener);
   }

   public static final int DEFAULT_WIDTH = 400;
   public static final int DEFAULT_HEIGHT = 200;

   private JPanel buttonPanel;
   private ButtonGroup group;
```

```
    private JLabel label;

    private static final int DEFAULT_SIZE = 12;
}
```

API javax.swing.JRadioButton 1.2

- `JRadioButton(String label, Icon icon)`

  构造一个单选按钮，初始没有被选择。
- `JRadioButton(String label, boolean state)`

  用给定的标签和初始状态构造一个单选按钮。

API javax.swing.ButtonGroup 1.2

- `void add(AbstractButton b)`

  将按钮添加到组中。
- `ButtonModel getSelection()`

  返回被选择的按钮的按钮模型。

API javax.swing.ButtonModel 1.2

- `String getActionCommand()`

  返回按钮模型的动作命令。

API javax.swing.AbstractButton 1.2

- `void setActionCommand(String s)`

  设置按钮及其模型的动作命令。

9.4.3 边框

如果在一个窗口中有多组单选按钮，就需要用可视化的形式指明哪些按钮属于同一组。Swing提供了一组很有用的边框（borders）来解决这个问题。可以在任何继承了JComponent的组件上应用边框。最常用的用途是在一个面板周围放置一个边框，然后用其他用户界面元素（如单选按钮）填充面板。

有几种不同的边框可供选择，但是使用它们的步骤完全一样。

1）调用BorderFactory的静态方法创建边框。下面是几种可选的风格（如图9-16所示）：

- 凹斜面
- 凸斜面
- 蚀刻
- 直线
- 不光滑

2）如果愿意的话，可以给边框添加标题，具体的实现方法是将边框传递给：

```
BorderFactory.createTitledBorder.
```

3）如果确实想把一切凸显出来，可以调用下列方法将几种边框组合起来使用：

```
BorderFactory.createCompoundBorder.
```

4）调用JComponent类中setBorder方法将结果边框添加到组件中。

例如，下面代码说明了如何把一个带有标题的蚀刻边框添加到一个面板上：

```
Border etched = BorderFactory.createEtchedBorder()
Border titled = BorderFactory.createTitledBorder(etched, "A Title");
panel.setBorder(titled);
```

运行例9-8中的程序可以看到各种边框的外观。

不同的边框有不同的用于设置边框的宽度和颜色的选项。详情请参看API注释。偏爱使用边框的人都很欣赏这一点，SoftBevelBorder类用于构造具有柔和拐角的斜面边框，LineBorder类也能够构造圆拐角。这些边框只能通过类中的某个构造器构造，而没有BorderFactory方法。

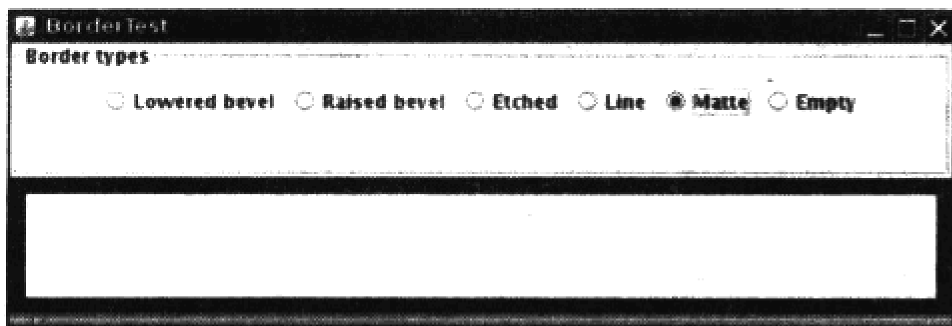


图9-16 测试边框类型

例9-5 BorderTest.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;

/**
 * @version 1.33 2007-06-12
 * @author Cay Horstmann
 */
public class BorderTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               BorderFrame frame = new BorderFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

```

```
/**
 * A frame with radio buttons to pick a border style.
 */
class BorderFrame extends JFrame
{
   public BorderFrame()
   {
      setTitle("BorderTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      demoPanel = new JPanel();
      buttonPanel = new JPanel();
      group = new ButtonGroup();

      addRadioButton("Lowered bevel", BorderFactory.createLoweredBevelBorder());
      addRadioButton("Raised bevel", BorderFactory.createRaisedBevelBorder());
      addRadioButton("Etched", BorderFactory.createEtchedBorder());
      addRadioButton("Line", BorderFactory.createLineBorder(Color.BLUE));
      addRadioButton("Matte", BorderFactory.createMatteBorder(10, 10, 10, 10, Color.BLUE));
      addRadioButton("Empty", BorderFactory.createEmptyBorder());

      Border etched = BorderFactory.createEtchedBorder();
      Border titled = BorderFactory.createTitledBorder(etched, "Border types");
      buttonPanel.setBorder(titled);

      setLayout(new GridLayout(2, 1));
      add(buttonPanel);
      add(demoPanel);
   }

   public void addRadioButton(String buttonName, final Border b)
   {
      JRadioButton button = new JRadioButton(buttonName);
      button.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               demoPanel.setBorder(b);
            }
         });
      group.add(button);
      buttonPanel.add(button);
   }

   public static final int DEFAULT_WIDTH = 600;
   public static final int DEFAULT_HEIGHT = 200;

   private JPanel demoPanel;
   private JPanel buttonPanel;
   private ButtonGroup group;
}
```



javax.swing.BorderFactory 1.2

- static Border createLineBorder(Color color)

- static Border createLineBorder(Color color, int thickness)

  创建一个简单的直线边框。
- static MatteBorder createMatteBorder(int top, int left, int bottom, int right, Color color)
- static MatteBorder createMatteBorder(int top, int left, int bottom, int right, Icon tileIcon)

  创建一个用color颜色或一个重复（repeating）图标填充的粗的边框。
- static Border createEmptyBorder()
- static Border createEmptyBorder(int top, int left, int bottom, int right)

  创建一个空边框。
- static Border createEtchedBorder()
- static Border createEtchedBorder(Color highlight, Color shadow)
- static Border createEtchedBorder(int type)
- static Border createEtchedBorder(int type, Color highlight, Color shadow)

  创建一个具有3D效果的直线边框。

  参数：highlight, shadow　用于 3D 效果的颜色

  　　　type　EtchedBorder.RAISED和 EtchedBorder.LOWERED之一
- static Border createBevelBorder (int type)
- static Border createBevelBorder(int type, Color highlight, Color shadow)
- static Border createLoweredBevelBorder()
- static Border createRaisedBevelBorder()

  创建一个具有凹面或凸面效果的边框。

  参数：type　BevelBorder.LOWERED和 BevelBorder.RAISED之一

  　　　highlight, shadow　用于3D效果的颜色
- static TitledBorder createTitledBorder(String title)
- static TitledBorder createTitledBorder(Border border)
- static TitledBorder createTitledBorder(Border border, String title)
- static TitledBorder createTitledBorder(Border border, String title, int justification, int position)
- static TitledBorder createTitledBorder(Border border, String title, int justification, int position, Font font)
- static TitledBorder createTitledBorder(Border border, String title, int justification, int position, Font font, Color color)

  创建一个具有给定特性的带标题的边框。

  参数：title　标题字符串

  　　　border　用标题装饰的边框

  　　　justification　TitledBorder 常量LEFT、CENTER、RIGHT、LEADING、

	trAILING或 DEFAULT_JUSTIFICATION (left) 之一
position	TitledBorder常量ABOVE_TOP、TOP、BELOW_TOP、ABOVE_BOTTOM、BOTTOM、BELOW_BOTTOM或DEFAULT_POSITION (top)之一
font	标题的字体
color	标题的颜色

- `static CompoundBorder createCompoundBorder(Border outsideBorder, Border insideBorder)`

  将两个边框组合成一个新的边框。

API javax.swing.border.SoftBevelBorder 1.2

- `SoftBevelBorder(int type)`
- `SoftBevelBorder(int type, Color highlight, Color shadow)`

  创建一个带有柔和角的斜面边框。

  参数：type　　BevelBorder.LOWERED和BevelBorder.RAISED之一

  highlight, shadow　　用于3D效果的颜色

API javax.swing.border.LineBorder 1.2

- `public LineBorder(Color color, int thickness, boolean roundedCorners)`

  用指定的颜色和宽度创建一个直线边框。如果 roundedCorners 为true，则边框具有圆拐角。

API javax.swing.JComponent 1.2

- `void setBorder(Border border)`

  设置这个组件的边框。

9.4.4 组合框

如果有多个选择项，使用单选按钮就不太适宜了，其原因是占据的屏幕空间太大。这时就可以选择组合框。当用户点击这个组件时，选择列表就会下拉出来，用户可以从中选择一项（见图9-17）。

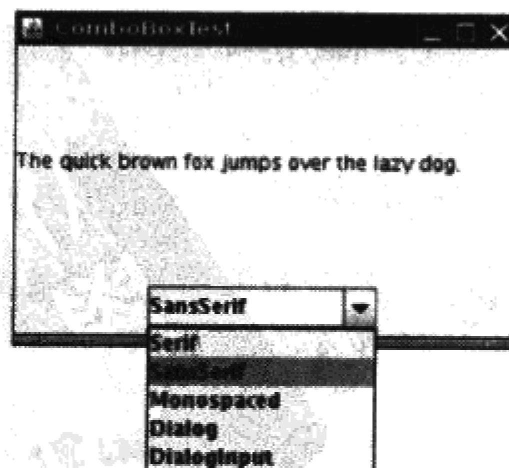


图9-17 组合框

如果下拉列表框被设置成可编辑（editable），就可以像编辑文本一样编辑当前的选项内容。鉴于这个原因，这种组件被称为组合框（combo box），它将文本域的灵活性与一组预定义的选项组合起来。JComboBox类提供了组合框的组件。

调用setEditable方法可以让组合框可编辑。注意，编辑只会影响当前项，而不会改变列表内容。

可以调用getSelectedItem方法获取当前的选项或被编辑的文本。在示例程序中，用户可以从

字体列表（Serif, SansSerif, Monospaced等）中选择一种字体，用户也可以键入其他的字体。

可以调用addItem方法增加选项。在示例程序中，只在构造器中调用了addItem方法，实际上，可以在任何地方调用它。

```
faceCombo = new JComboBox();
faceCombo.setEditable(true);
faceCombo.addItem("Serif");
faceCombo.addItem("SansSerif");
. . .
```

这个方法将字符串添加到列表的尾部。可以利用insertItemAt方法在列表的任何位置插入一个新选项：

```
faceCombo.insertItemAt("Monospaced", 0); // add at the beginning
```

可以增加任何类型的选项，组合框可以调用每个选项的toString方法显示其内容。

如果需要在运行时删除某些选项，可以使用removeItem 或者 removeItemAt 方法，使用哪个方法将取决于参数提供的是想要删除的选项内容，还是选项位置。

```
faceCombo.removeItem("Monospaced");
faceCombo.removeItemAt(0); // remove first item
```

调用removeAllItems方法将立即移除所有的选项。

提示：如果需要往组合框中添加大量的选项，addItem方法的性能就显得很差了。取而代之的是构造一个DefaultComboBoxModel，并调用addElement方法进行加载，然后再调用JComboBox中的setModel方法。

当用户从组合框中选择一个选项时，组合框就将产生一个动作事件。为了判断哪个选项被选择，可以通过事件参数调用getSource方法来得到发送事件的组合框引用，接着调用getSelectedItem方法获取当前选择的选项。需要把这个方法的返回值转化为相应的类型，通常是String型。

```
public void actionPerformed(ActionEvent event)
{
   label.setFont(new Font(
      (String) faceCombo.getSelectedItem(),
      Font.PLAIN,
      DEFAULT_SIZE));
}
```

例9-6给出了完整的代码。

注释：如果希望持久地显示列表，而不是下拉列表，就应该使用JList组件。在卷II的第6章中将介绍JList。

例9-6 ComboBoxTest.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

```

```
/**
 * @version 1.33 2007-06-12
 * @author Cay Horstmann
 */
public class ComboBoxTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {

               ComboBoxFrame frame = new ComboBoxFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * A frame with a sample text label and a combo box for selecting font faces.
 */
class ComboBoxFrame extends JFrame
{
   public ComboBoxFrame()
   {
      setTitle("ComboBoxTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // add the sample text label

      label = new JLabel("The quick brown fox jumps over the lazy dog.");
      label.setFont(new Font("Serif", Font.PLAIN, DEFAULT_SIZE));
      add(label, BorderLayout.CENTER);

      // make a combo box and add face names

      faceCombo = new JComboBox();
      faceCombo.setEditable(true);
      faceCombo.addItem("Serif");
      faceCombo.addItem("SansSerif");
      faceCombo.addItem("Monospaced");
      faceCombo.addItem("Dialog");
      faceCombo.addItem("DialogInput");

      // the combo box listener changes the label font to the selected face name

      faceCombo.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               label.setFont(new Font((String) faceCombo.getSelectedItem(), Font.PLAIN,
                     DEFAULT_SIZE));
            }
         });
```

```

        // add combo box to a panel at the frame's southern border

        JPanel comboPanel = new JPanel();
        comboPanel.add(faceCombo);
        add(comboPanel, BorderLayout.SOUTH);
    }

    public static final int DEFAULT_WIDTH = 300;
    public static final int DEFAULT_HEIGHT = 200;

    private JComboBox faceCombo;
    private JLabel label;
    private static final int DEFAULT_SIZE = 12;
}
```

API javax.swing.JComboBox 1.2

- `boolean isEditable()`
- `void setEditable(boolean b)`

  获取或设置组合框的可编辑特性。
- `void addItem(Object item)`

  把一个选项添加到选项列表中。
- `void insertItemAt(Object item, int index)`

  将一个选项添加到选项列表的指定位置。
- `void removeItem(Object item)`

  从选项列表中删除一个选项。
- `void removeItemAt(int index)`

  删除指定位置的选项。
- `void removeAllItems()`

  从选项列表中删除所有选项。
- `Object getSelectedItem()`

  返回当前选择的选项。

9.4.5 滑块

组合框可以让用户从一组离散值中进行选择。滑块允许进行连续值得选择，例如，从1～100之间选择任意数值。

通常，可以使用下列方式构造滑块：

```
JSlider slider = new JSlider(min, max, initialValue);
```

如果省略最小值、最大值和初始值，其默认值分别为0、100和50。

如果需要垂直滑块，可以按照下列方式调用构造器：

```
JSlider slider = new JSlider(SwingConstants.VERTICAL, min, max, initialValue);
```

这个构造器构造了一个无格式的滑块，如同图9-18最上面的滑块所示。下面看一下如何为滑块添加装饰。

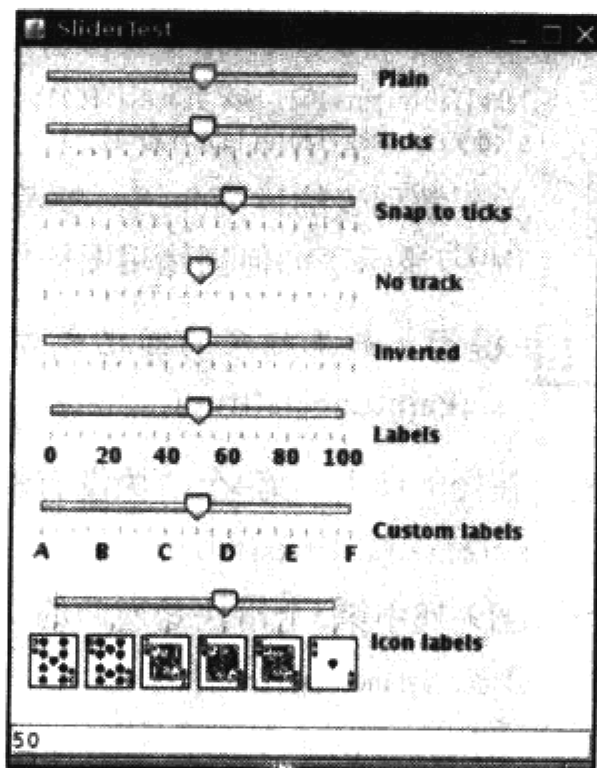


图9-18 滑块

当用户滑动滑块时，滑块的值就会在最小值和最大值之间变化。当值发生变化时，ChangeEvent就会发送给所有变化的监听器。为了得到这些改变的通知，需要调用addChangeListener方法并且安装一个实现了ChangeListener接口的对象。这个接口只有一个方法StateChanged。在这个方法中，可以获取滑块的当前值：

```
public void stateChanged(ChangeEvent event)
{
   JSlider slider = (JSlider) event.getSource();
   int value = slider.getValue();
   . . .
}
```

可以通过显示标尺（tricks）对滑块进行修饰。例如，在示例程序中，第二个滑块使用了下面的设置：

```
slider.setMajorTickSpacing(20);
slider.setMinorTickSpacing(5);
```

上述滑块在每20个单元的位置显示一个大标尺标记，每5个单元的职位显示一个小标尺标记。所谓单元是指滑块值，而不是像素。

这些代码只设置了标尺标记，要想将它们显示出来，还需要调用：

```
slider.setPaintTicks(true);
```

大标尺和小标尺是相互独立的。例如，可以每20个单元设置一个大标尺，同时每7个单元设置一个小标尺，但是这样设置，滑块看起来会显得非常凌乱。

可以强制滑块对齐标尺。这样一来，只要用户完成拖放滑块的操作，滑块就会立即自动地移到最接近的标尺处。激活这种操作方式需要调用：

```
slider.setSnapToTicks(true);
```

注释：“对齐标尺”的行为与想像的工作过程并不太一样。在滑块真正对齐之前，改变监听器报告的滑块值并不是对应的标尺值。如果点击了滑块附近，滑块将会向点击的方向移动一小段距离，“对齐标尺”的滑块并不移动到下一个标尺处。

可以调用下列方法为大标尺添加标尺标签(tick mark labels)：

```
slider.setPaintLabels(true);
```

例如，对于一个范围为0到100的滑块，如果大标尺的间距是20，每个大标尺的标签就应该分别是0、20、40、60、80和100。

还可以提供其他形式的标尺标记，如字符串或者图标（见图9-18）。这样做有些烦琐。首先需要填充一个键为Integer类型且值为Component类型的散列表（在JDK5.0中自打包让这个过程变得十分容易）。然后再调用setLabelTable方法，组件就会放置在标尺标记处。通常组件使用的是JLabel对象。下面代码说明了如何将标尺标签设置为A、B、C、D、E和F。

```
Hashtable<Integer, Component> labelTable = new Hashtable<Integer, Component>();
labelTable.put(0, new JLabel("A"));
labelTable.put(20, new JLabel("B"));
. . .
labelTable.put(100, new JLabel("F"));
slider.setLabelTable(labelTable);
```

关于散列表的详细介绍，参考卷II的第2章。

例9-7显示了如何创建用图标作为标尺标签的滑块。

提示：如果标尺的标记或者标签不显示，请检查一下是否调用了setPaintTicks(true)和setPaintLabels(true)。

在图9-18中，第4个滑块没有轨迹。要想隐藏滑块移动的轨迹，可以调用：

```
slider.setPaintTrack(false);
```

图9-18中第5个滑块是逆向的，调用下列方法可以实现这个效果：

```
slider.setInverted(true);
```

示例程序演示了所有不同视觉效果的滑块。每个滑块都安装了一个改变事件监听器，它负责把当前的滑块值显示到框架底部的文本域中。

例9-7 SliderTest.java

```
import java.awt.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;

/**
 * @version 1.13 2007-06-12
 * @author Cay Horstmann
 */
public class SliderTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               SliderTestFrame frame = new SliderTestFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * A frame with many sliders and a text field to show slider values.
 */
class SliderTestFrame extends JFrame
{
   public SliderTestFrame()
```

```
   {
      setTitle("SliderTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      sliderPanel = new JPanel();
      sliderPanel.setLayout(new FlowLayout(FlowLayout.LEFT));

      // common listener for all sliders
      listener = new ChangeListener()
         {
            public void stateChanged(ChangeEvent event)
            {
               // update text field when the slider value changes
               JSlider source = (JSlider) event.getSource();
               textField.setText("" + source.getValue());
            }
         };

      // add a plain slider

      JSlider slider = new JSlider();
      addSlider(slider, "Plain");

      // add a slider with major and minor ticks

      slider = new JSlider();
      slider.setPaintTicks(true);
      slider.setMajorTickSpacing(20);
      slider.setMinorTickSpacing(5);
      addSlider(slider, "Ticks");

      // add a slider that snaps to ticks

      slider = new JSlider();
      slider.setPaintTicks(true);
      slider.setSnapToTicks(true);
      slider.setMajorTickSpacing(20);
      slider.setMinorTickSpacing(5);
      addSlider(slider, "Snap to ticks");

      // add a slider with no track

      slider = new JSlider();
      slider.setPaintTicks(true);
      slider.setMajorTickSpacing(20);
      slider.setMinorTickSpacing(5);
      slider.setPaintTrack(false);
      addSlider(slider, "No track");

      // add an inverted slider

      slider = new JSlider();
      slider.setPaintTicks(true);
      slider.setMajorTickSpacing(20);
      slider.setMinorTickSpacing(5);
      slider.setInverted(true);
      addSlider(slider, "Inverted");
```

```
      // add a slider with numeric labels

      slider = new JSlider();
      slider.setPaintTicks(true);
      slider.setPaintLabels(true);
      slider.setMajorTickSpacing(20);
      slider.setMinorTickSpacing(5);
      addSlider(slider, "Labels");

      // add a slider with alphabetic labels

      slider = new JSlider();
      slider.setPaintLabels(true);
      slider.setPaintTicks(true);
      slider.setMajorTickSpacing(20);
      slider.setMinorTickSpacing(5);

      Dictionary<Integer, Component> labelTable = new Hashtable<Integer, Component>();
      labelTable.put(0, new JLabel("A"));
      labelTable.put(20, new JLabel("B"));
      labelTable.put(40, new JLabel("C"));
      labelTable.put(60, new JLabel("D"));
      labelTable.put(80, new JLabel("E"));
      labelTable.put(100, new JLabel("F"));

      slider.setLabelTable(labelTable);
      addSlider(slider, "Custom labels");

      // add a slider with icon labels

      slider = new JSlider();
      slider.setPaintTicks(true);
      slider.setPaintLabels(true);
      slider.setSnapToTicks(true);
      slider.setMajorTickSpacing(20);
      slider.setMinorTickSpacing(20);

      labelTable = new Hashtable<Integer, Component>();

      // add card images

      labelTable.put(0, new JLabel(new ImageIcon("nine.gif")));
      labelTable.put(20, new JLabel(new ImageIcon("ten.gif")));
      labelTable.put(40, new JLabel(new ImageIcon("jack.gif")));
      labelTable.put(60, new JLabel(new ImageIcon("queen.gif")));
      labelTable.put(80, new JLabel(new ImageIcon("king.gif")));
      labelTable.put(100, new JLabel(new ImageIcon("ace.gif")));

      slider.setLabelTable(labelTable);
      addSlider(slider, "Icon labels");

      // add the text field that displays the slider value

      textField = new JTextField();
      add(sliderPanel, BorderLayout.CENTER);
      add(textField, BorderLayout.SOUTH);
```

```
    }

    /**
     * Adds a slider to the slider panel and hooks up the listener
     * @param s the slider
     * @param description the slider description
     */
    public void addSlider(JSlider s, String description)
    {
       s.addChangeListener(listener);
       JPanel panel = new JPanel();
       panel.add(s);
       panel.add(new JLabel(description));
       sliderPanel.add(panel);
    }

    public static final int DEFAULT_WIDTH = 350;
    public static final int DEFAULT_HEIGHT = 450;

    private JPanel sliderPanel;
    private JTextField textField;
    private ChangeListener listener;
}
```

API javax.swing.JSlider 1.2

- JSlider()
- JSlider(int direction)
- JSlider(int min, int max)
- JSlider(int min, int max, int initialValue)
- JSlider(int direction, int min, int max, int initialValue)

  用给定的方向、最大值、最小值和初始化值构造一个水平滑块。

参数：	
direction	SwingConstants.HORIZONTAL 或SwingConstants.VERTICAL之一。默认为水平
min, max	滑块的最大值、最小值。默认值为0到100
initialValue	滑块的初始化值。默认值为50

- void setPaintTicks(boolean b)

  如果b为true，显示标尺。

- void setMajorTickSpacing(int units)
- void setMinorTickSpacing(int units)

  用给定的滑块单元的倍数设置大标尺和小标尺。

- void setPaintLabels(boolean b)

  如果b是true，显示标尺标签。

- void setLabelTable(Dictionary table)

  设置用于作为标尺标签的组件。表中的每一个键/值对都采用new Integer(value)/component

的格式。

- void setSnapToTicks(boolean b)

  如果b是true，每一次调整后滑块都要对齐到最接近的标尺处。

- void setPaintTrack(boolean b)

  如果b是true，显示滑块滑动的轨迹。

9.5 菜单

前面介绍了几种最常用的可以放到窗口内的组件，如：各种按钮、文本域以及组合框等。Swing还提供了一些其他种类的用户界面元素，下拉式菜单就是GUI应用程序中最常见的一种。

位于窗口顶部的*菜单栏*（menu bar）包括了下拉菜单的名字。点击一个名字就可以打开包含*菜单项*（menu items）和*子菜单*（submenus）的菜单。当用户点击菜单项时，所有的菜单都会被关闭并且将一条消息发送给程序。图9-19显示了一个带子菜单的典型菜单。

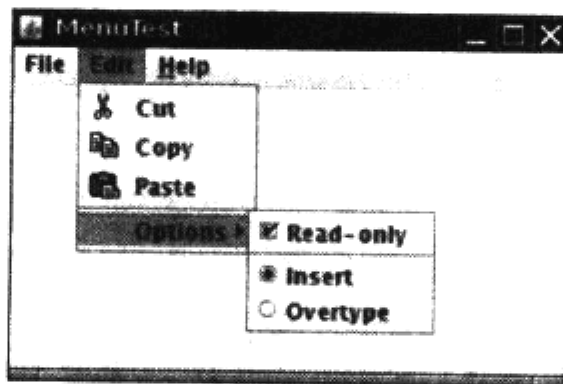


图9-19 带有子菜单的菜单

9.5.1 菜单创建

创建菜单是一件非常容易的事情。首先要创建一个菜单栏：

```
JMenuBar menuBar = new JMenuBar();
```

菜单栏是一个可以添加到任何位置的组件。通常放置在框架的顶部。可以调用setJMenuBar方法将菜单栏添加到框架上：

```
frame.setJMenuBar(menuBar);
```

需要为每个菜单建立一个菜单对象：

```
JMenu editMenu = new JMenu("Edit");
```

然后将顶层菜单添加到菜单栏中：

```
menuBar.add(editMenu);
```

向菜单对象中添加菜单项、分隔符和子菜单：

```
JMenuItem pasteItem = new JMenuItem("Paste");
editMenu.add(pasteItem);
editMenu.addSeparator();
JMenu optionsMenu = . . .; // a submenu
editMenu.add(optionsMenu);
```

可以看到分隔符位于Paste和Read-only菜单项之间。

当用户选择菜单时，将触发一个动作事件。这里需要为每个菜单项安装一个动作监听器。

```
ActionListener listener = . . .;
pasteItem.addActionListener(listener);
```

可以使用JMenu.add(String s)方法将菜单项插入到菜单的尾部，例如：

```
editMenu.add("Paste");
```

Add方法返回创建的子菜单项。可以采用下列方式获取它，并添加监听器：

```
JMenuItem pasteItem = editMenu.add("Paste");
pasteItem.addActionListener(listener);
```

在通常情况下，菜单项触发的命令也可以通过其他用户界面元素（如工具栏上的按钮）激活。在第8章中，已经看到了如何通过Action对象来指定命令。通常，采用扩展抽象类AbstractAction来定义一个实现Action接口的类。这里需要在AbstractAction对象的构造器中指定菜单项标签并且覆盖actionPerformed方法来获得菜单动作处理器。例如：

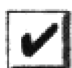
```
Action exitAction = new AbstractAction("Exit") // menu item text goes here
   {
      public void actionPerformed(ActionEvent event)
      {
         // action code goes here
         System.exit(0);
      }
   };
```

然后将动作添加到菜单中：

```
JMenuItem exitItem = fileMenu.add(exitAction);
```

这个命令利用动作名将一个菜单项添加到菜单中。这个动作对象将作为它的监听器。上面这条语句是下面两条语句的缩写形式：

```
JMenuItem exitItem = new JMenuItem(exitAction);
fileMenu.add(exitItem);
```



注释：*在Windows和Macintosh程序中，通常菜单定义在外部资源文件中，并利用资源标识符将其绑定到应用程序。在Java中，菜单通常在程序内部创建，这是因为Java处理外部资源的机制比Windows和Mac操作系统的限制更多。*



javax.swing.JMenu 1.2

- `JMenu(String label)`

  用给定标签构造一个菜单。
- `JMenuItem add(JMenuItem item)`

  添加一个菜单项（或一个菜单）。
- `JMenuItem add(String label)`

  用给定标签将一个菜单项添加到菜单中，并返回这个菜单项。
- `JMenuItem add(Action a)`

  用给定动作将一个菜单项添加到菜单中，并返回这个菜单项。
- `void addSeparator()`

  将一个分隔符行（separator line）添加到菜单中。
- `JMenuItem insert(JMenuItem menu, int index)`

将一个新菜单项（或子菜单）添加到菜单的指定位置。

- JMenuItem insert(Action a, int index)

  用给定动作在菜单的指定位置添加一个新菜单项。

- void insertSeparator(int index)

  将一个分隔符添加到菜单中。

  参数：index　　添加分隔符的位置

- void remove(int index)
- void remove(JMenuItem item)

  从菜单中删除指定的菜单项。

API javax.swing.JMenuItem 1.2

- JMenuItem(String label)

  用给定标签构造一个菜单项。

- JMenuItem(Action a) 1.3

  为给定动作构造一个菜单项。

API javax.swing.AbstractButton 1.2

- void setAction(Action a) 1.3

  为这个按钮或菜单项设置动作。

API javax.swing.JFrame 1.2

- void setJMenuBar(JMenuBar menubar)

  为这个框架设置菜单栏。

9.5.2 菜单项中的图标

菜单项与按钮很相似。实际上，JMenuItem类扩展了AbstractButton类。与按钮一样，菜单可以包含文本标签、图标，也可以两者都包含。既可以利用JMenuItem(String, Icon)或者JMenuItem(Icon) 构造器为菜单指定一个图标，也可以利用JMenuItem类中的setIcon方法（继承自AbstractButton类）指定一个图标。例如：

```
JMenuItem cutItem = new JMenuItem("Cut", new ImageIcon("cut.gif"));
```

图9-19展示了具有图标的菜单项。正如所看到的，在默认情况下，菜单项的文本被放置在图标的右侧。如果喜欢将文本放置在左侧，可以调用JMenuItem类中的setHorizontalTextPosition方法（继承自AbstractButton类）设置。例如：

```
cutItem.setHorizontalTextPosition(SwingConstants.LEFT);
```

这个调用把菜单项文本移动到图标的左侧。

也可以将一个图标添加到一个动作上：

```
cutAction.putValue(Action.SMALL_ICON, new ImageIcon("cut.gif"));
```

当使用动作构造菜单项时，Action.NAME值将会作为菜单项的文本，而Action.SMALL_ICON将会作为图标。

另外，可以利用AbstractAction构造器设置图标：

```
cutAction = new
   AbstractAction("Cut", new ImageIcon("cut.gif"))
   {
      public void actionPerformed(ActionEvent event)
      {
         // action code goes here
      }
   };
```

API javax.swing.JMenuItem 1.2

- `JMenuItem(String label, Icon icon)`

  用给定的标签和图标构造一个菜单项。

API javax.swing.AbstractButton 1.2

- `void setHorizontalTextPosition(int pos)`

  设置文本对应图标的水平位置。

  参数：pos　SwingConstants.RIGHT（文本在图标的右侧）或 SwingConstants.LEFT

API javax.swing.AbstractAction 1.2

- `AbstractAction(String name, Icon smallIcon)`

  用给定的名字和图标构造一个抽象的动作。

9.5.3 复选框和单选按钮菜单项

*复选框*和*单选按钮*菜单项在文本旁边显示了一个复选框或一个单选按钮（参见图9-19）。当用户选择一个菜单项时，菜单项就会自动地在选择和未选择间进行切换。

除了按钮装饰外，同其他菜单项的处理一样。例如，下面是创建复选框菜单项的代码：

```
JCheckBoxMenuItem readonlyItem = new JCheckBoxMenuItem("Read-only");
optionsMenu.add(readonlyItem);
```

单选按钮菜单项与普通单选按钮的工作方式一样，必须将它们加入到按钮组中。当按钮组中的一个按钮被选中时，其他按钮都自动地变为未选择项。

```
ButtonGroup group = new ButtonGroup();
JRadioButtonMenuItem insertItem = new JRadioButtonMenuItem("Insert");
insertItem.setSelected(true);
JRadioButtonMenuItem overtypeItem = new JRadioButtonMenuItem("Overtype");
group.add(insertItem);
group.add(overtypeItem);
optionsMenu.add(insertItem);
optionsMenu.add(overtypeItem);
```

使用这些菜单项，不需要立刻得到用户选择菜单项的通知。而是使用isSelected方法来测试

菜单项的当前状态（当然，这意味着应该保留一个实例域保存这个菜单项的引用）。使用setSelect方法设置状态。

API javax.swing.JCheckBoxMenuItem 1.2

- `JCheckBoxMenuItem(String label)`
  用给定的标签构造一个复选框菜单项。
- `JCheckBoxMenuItem(String label, boolean state)`
  用给定的标签和给定的初始状态（true为选定）构造一个复选框菜单。

API javax.swing.JRadioButtonMenuItem 1.2

- `JRadioButtonMenuItem(String label)`
  用给定的标签构造一个单选按钮菜单项。
- `JRadioButtonMenuItem(String label, boolean state)`
  用给定的标签和给定的初始状态（true为选定）构造一个单选按钮菜单项。

API javax.swing.AbstractButton 1.2

- `boolean isSelected()`
- `void setSelected(boolean state)`
  获取或设置这个菜单项的选择状态（true为选定）。

9.5.4 弹出菜单

弹出菜单（pop-up menu）是不固定在菜单栏中随处浮动的菜单（参见图9-20）

创建一个弹出菜单与创建一个常规菜单的方法类似，但是弹出菜单没有标题。

```
JPopupMenu popup = new JPopupMenu();
```

然后用常规的方法添加菜单项：

```
JMenuItem item = new JMenuItem("Cut");
item.addActionListener(listener);
popup.add(item);
```

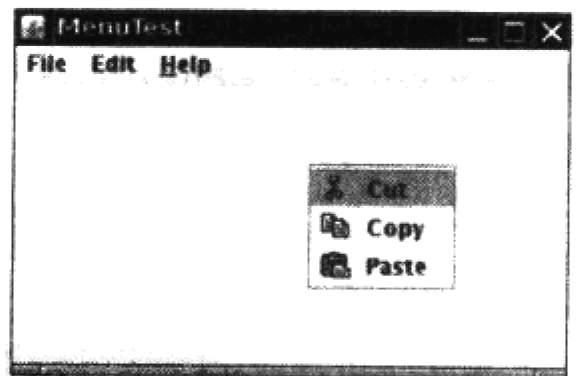


图9-20 弹出菜单

弹出菜单并不像常规菜单栏那样总是显示在框架的顶部，必须调用show方法菜单才能显示出来。调用时需要给出父组件以及相对父组件坐标的显示位置。例如：

```
popup.show(panel, x, y);
```

通常，当用户点击某个鼠标键时弹出菜单。这就是所谓的*弹出式触发器*（pop-up trigger）。在Windows或者Linux中，弹出式触发器是鼠标右键。要想在用户点击某一个组件时弹出菜单，需要按照下列方式调用方法：

```
component.setComponentPopupMenu(popup);
```

偶尔会遇到在一个含有弹出菜单的组件中放置一个组件的情况。这个子组件可以调用下列方法继承父组件的弹出菜单。调用：

```
child.setInheritsPopupMenu(true);
```

这些方法是SE5.0中新增加的，这样可以让程序员不必关心系统对弹出菜单的依赖。在SE5.0之前，必须安装鼠标监听器，并将下列代码添加到mousePressed和mouseReleased监听器中：

```
if (popup.isPopupTrigger(event))
   popup.show(event.getComponent(), event.getX(), event.getY());
```

有些系统在按钮压下时触发弹出菜单，有些系统则按钮抬起时触发。

API `javax.swing.JPopupMenu 1.2`

- `void show(Component c, int x, int y)`

  显示一个弹出菜单。

  参数：c　　显示弹出菜单的组件

  　　　x, y　　弹出菜单左上角的坐标（c的坐标空间内）

- `boolean isPopupTrigger(MouseEvent event) 1.3`

  如果鼠标事件是弹出菜单触发器，则返回 true。

API `java.awt.event.MouseEvent 1.1`

- `boolean isPopupTrigger()`

  如果鼠标事件是弹出菜单触发器，则返回 true 。

API `javax.swing.JComponent 1.2`

- `JPopupMenu getComponentPopupMenu() 5.0`
- `void setComponentPopupMenu(JPopupMenu popup) 5.0`

  获取或设置用于这个组件的弹出菜单。

- `boolean getInheritsPopupMenu() 5.0`
- `void setInheritsPopupMenu(boolean b) 5.0`

  获取或设置 inheritsPopupMenu特性。如果这个特性被设置或这个组件的弹出菜单为null，则应用上一级弹出菜单。

9.5.5 快捷键和加速器

对于有经验的用户来说，通过快捷键来选择菜单项会感觉更加便捷。可以通过在菜单项的构造器中指定一个快捷字母来为菜单项设置快捷键：

```
JMenuItem aboutItem = new JMenuItem("About", 'A');
```

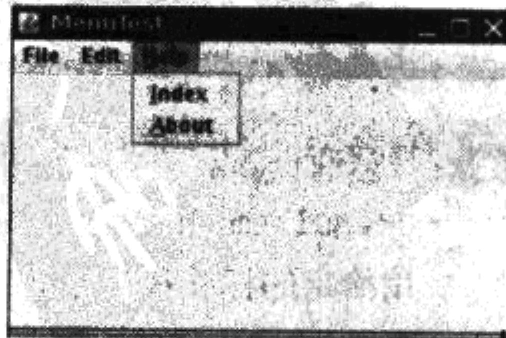


图9-21 键盘快捷键

快捷键会自动地显示在菜单项中，并带有一条下划线（如图9-21）。例如，在上面的例子中，菜单项中的标签为“About”，

字母A带有一个下划线。当显示菜单时，用户只需要按下“A”键就可以这个选择菜单项（如果快捷字母没有出现在菜单项标签字符串中，同样可以按下快捷键选择菜单项，只是快捷键没有显示出来。很自然，这种不可见的快捷键没有提示效果）。

有时候不希望在菜单项的第一个快捷键字母下面加下划线。例如，如果在菜单项“Save As”中使用快捷键“A”，则在第二个“A”（Save As）下面加下划线更为合理。在Java SE1.4中，可以调用setDisplayedMnemonicIndex方法指定希望加下划线的字符。

如果有一个Action对象，就可以把快捷键作为Action. MNEMONIC_KEY的键值添加到对象中。如：

```
cutAction.putValue(Action.MNEMONIC_KEY, new Integer('A'));
```

只能在菜单项的构造器中设定快捷键字母，而不是在菜单构造器中。如果想为菜单设置快捷键，需要调用setMnemonic方法：

```
JMenu helpMenu = new JMenu("Help");
helpMenu.setMnemonic('H');
```

可以同时按下ALT键和菜单的快捷键来实现在菜单栏中选择一个顶层菜单的操作。例如：按下ALT+H可以从菜单中选择Help菜单项。

可以使用快捷键从当前打开的菜单中选择一个子菜单或者菜单项。而加速器是在不打开菜单的情况下选择菜单项的快捷键。例如：很多程序把加速器CTRL+O和CTRL+S关联到File菜单中的Open和Save菜单项。可以使用setAccelerator将加速器键关联到一个菜单项上。这个方法使用KeyStroke类型的对象作为参数。例如：下面的调用将加速器CTRL+O关联到OpenItem菜单项。

```
openItem.setAccelerator(KeyStroke.getKeyStroke("ctrl O"));
```

当用户按下加速器组合键时，就会自动地选择相应的菜单项，同时激活一个动作事件，这与手工地选择这个菜单项一样。

加速器只能关联到菜单项上，不能关联到菜单上。加速器键并不实际打开菜单。它将直接地激活菜单关联的动作事件。

从概念上讲，把加速器添加到菜单项与把加速器添加到Swing组件上所使用的技术十分类似（在第8章中讨论了这个技术）。但是，当加速器添加到菜单项时，对应的组合键就会自动地显示在相应的菜单上（如图9-22）。

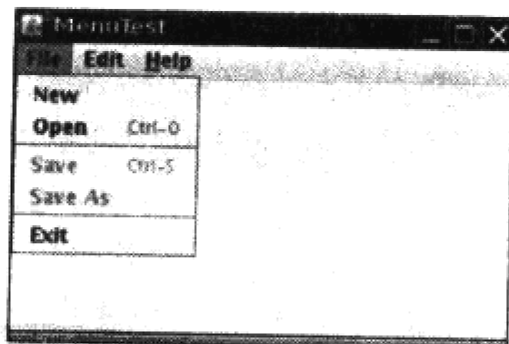


图9-22 加速键

> **注释**：在Windows下，ALT+F4用于关闭窗口。但这不是Java程序设定的加速键。这是操作系统定义的快捷键。这个组合键总会触发活动窗口的WindowClosing事件，而不管菜单上是否有Close菜单项。

API javax.swing.JMenuItem 1.2

- `JMenuItem(String label, int mnemonic)`

  用给定的标签和快捷键字符构造一个菜单项

  参数：label　菜单项的标签

  mnemonic　菜单项的快捷键字符，在标签中这个字符下面会有一个下划线。

- void setAccelerator(KeyStroke k)

  将k设置为这个菜单项的加速器。加速器显示在标签的旁边。

API javax.swing.AbstractButton 1.2

- void setMnemonic(int mnemonic)

  设置按钮的快捷字符。该字符会在标签中以下划线的形式显示。

  参数：mnemonic　按钮的快捷字符。

- void setDisplayedMnemonicIndex(int index) 1.4

  将按钮文本中的index字符设定为带下划线。如果不希望第一个出现的快捷键字符带下划线，就可以使用这个方法。

  参数：index　　在按钮文本中设定为带下划线的字符索引。

9.5.6 启用和禁用菜单项

在有些时候，某个特定的菜单项可能只能够在某种特定的环境下才可用。例如，当文档以只读方式打开时，Save菜单项就没有意义。当然，可以使用JMenu.remove方法将这个菜单项从菜单中删掉，但用户会对菜单内容的不断变化感到奇怪。然而，可以将这个菜单项设为禁用状态，以便屏蔽掉这些暂时不适用的命令。被禁用的菜单项被显示为灰色，不能被选择它（如图9-23）。

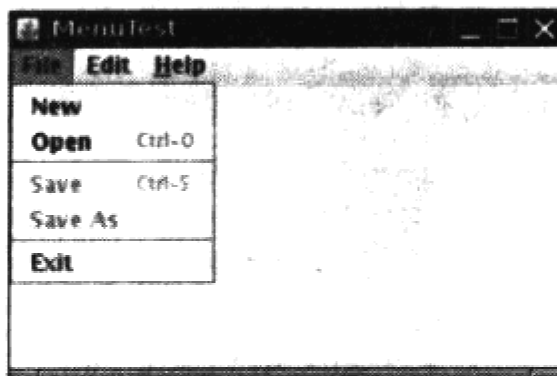


图9-23　禁用菜单项

启用或禁用菜单项需要调用setEnabled方法：

```
saveItem.setEnabled(false);
```

启用和禁用菜单项有两种策略。每次环境发生变化就对相关的菜单项或动作调用setEnabled。例如：只要当文档以只读方式打开，就禁用Save和Save As菜单项。另一种方法是在显示菜单之前禁用这些菜单项。这里必须为“menu selected”事件注册监听器。javax.swing.event包定义了MenuListener接口，它包含三个方法：

```
void menuSelected(MenuEvent event)
void menuDeselected(MenuEvent event)
void menuCanceled(MenuEvent event)
```

由于在菜单显示之前调用menuSelected方法，所以可以在这个方法中禁用或启用菜单项。下面代码显示了只读复选框菜单项被选择以后，如何禁用Save和Save As动作。

```
public void menuSelected(MenuEvent event)
{
   saveAction.setEnabled(!readonlyItem.isSelected());
   saveAsAction.setEnabled(!readonlyItem.isSelected());
}
```

警告：在显示菜单之前禁用菜单项是一种明智的选择，但这种方式不适用于带有加速键的菜单项。这是因为在敲击加速键时并没有打开菜单，因此动作没有被禁用，致使加速键还会触发这个行为。

API javax.swing.JMenuItem 1.2

- void setEnabled(boolean b)

  启用或禁用菜单项。

API javax.swing.event.MenuListener 1.20

- void menuSelected(MenuEvent e)

  在菜单被选择但尚未打开之前调用。

- void menuDeselected(MenuEvent e)

  在菜单被取消选择并且已经关闭之后被调用。

- void menuCanceled(MenuEvent e)

  当菜单被取消时被调用。例如，用户点击菜单以外的区域。

例9-8是创建一组菜单的示例程序。这个程序演示了本节介绍的所有特性，包括：嵌套菜单、禁用菜单项、复选框和单选按钮菜单项、弹出菜单以及快捷键和加速器。

例9-8 MenuTest.java

```
import java.awt.EventQueue;
import java.awt.event.*;
import javax.swing.*;

/**
 * @version 1.23 2007-05-30
 * @author Cay Horstmann
 */
public class MenuTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               MenuFrame frame = new MenuFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * A frame with a sample menu bar.
 */
class MenuFrame extends JFrame
{
   public MenuFrame()
   {
      setTitle("MenuTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
```

```
      JMenu fileMenu = new JMenu("File");
      fileMenu.add(new TestAction("New"));

      // demonstrate accelerators

      JMenuItem openItem = fileMenu.add(new TestAction("Open"));
      openItem.setAccelerator(KeyStroke.getKeyStroke("ctrl O"));

      fileMenu.addSeparator();

      saveAction = new TestAction("Save");
      JMenuItem saveItem = fileMenu.add(saveAction);
      saveItem.setAccelerator(KeyStroke.getKeyStroke("ctrl S"));

      saveAsAction = new TestAction("Save As");
      fileMenu.add(saveAsAction);
      fileMenu.addSeparator();

      fileMenu.add(new AbstractAction("Exit")
         {
            public void actionPerformed(ActionEvent event)
            {
               System.exit(0);
            }
         });

      // demonstrate check box and radio button menus

      readonlyItem = new JCheckBoxMenuItem("Read-only");
      readonlyItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               boolean saveOk = !readonlyItem.isSelected();
               saveAction.setEnabled(saveOk);
               saveAsAction.setEnabled(saveOk);
            }
         });

      ButtonGroup group = new ButtonGroup();

      JRadioButtonMenuItem insertItem = new JRadioButtonMenuItem("Insert");
      insertItem.setSelected(true);
      JRadioButtonMenuItem overtypeItem = new JRadioButtonMenuItem("Overtype");

      group.add(insertItem);
      group.add(overtypeItem);

      // demonstrate icons

      Action cutAction = new TestAction("Cut");
      cutAction.putValue(Action.SMALL_ICON, new ImageIcon("cut.gif"));
      Action copyAction = new TestAction("Copy");
      copyAction.putValue(Action.SMALL_ICON, new ImageIcon("copy.gif"));
      Action pasteAction = new TestAction("Paste");
      pasteAction.putValue(Action.SMALL_ICON, new ImageIcon("paste.gif"));
```

```
      JMenu editMenu = new JMenu("Edit");
      editMenu.add(cutAction);
      editMenu.add(copyAction);
      editMenu.add(pasteAction);

      // demonstrate nested menus

      JMenu optionMenu = new JMenu("Options");

      optionMenu.add(readonlyItem);
      optionMenu.addSeparator();
      optionMenu.add(insertItem);
      optionMenu.add(overtypeItem);

      editMenu.addSeparator();
      editMenu.add(optionMenu);

      // demonstrate mnemonics

      JMenu helpMenu = new JMenu("Help");
      helpMenu.setMnemonic('H');

      JMenuItem indexItem = new JMenuItem("Index");
      indexItem.setMnemonic('I');
      helpMenu.add(indexItem);

      // you can also add the mnemonic key to an action
      Action aboutAction = new TestAction("About");
      aboutAction.putValue(Action.MNEMONIC_KEY, new Integer('A'));
      helpMenu.add(aboutAction);

      // add all top-level menus to menu bar

      JMenuBar menuBar = new JMenuBar();
      setJMenuBar(menuBar);

      menuBar.add(fileMenu);
      menuBar.add(editMenu);
      menuBar.add(helpMenu);

      // demonstrate pop-ups

      popup = new JPopupMenu();
      popup.add(cutAction);
      popup.add(copyAction);
      popup.add(pasteAction);

      JPanel panel = new JPanel();
      panel.setComponentPopupMenu(popup);
      add(panel);

      // the following line is a workaround for bug 4966109
      panel.addMouseListener(new MouseAdapter()
         {
         });
   }
```

```

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;

   private Action saveAction;
   private Action saveAsAction;
   private JCheckBoxMenuItem readonlyItem;
   private JPopupMenu popup;
}

/**
 * A sample action that prints the action name to System.out
 */
class TestAction extends AbstractAction
{
   public TestAction(String name)
   {
      super(name);
   }

   public void actionPerformed(ActionEvent event)
   {
      System.out.println(getValue(Action.NAME) + " selected.");
   }
}
```

9.5.7 工具栏

工具栏是在程序中提供的快速访问常用命令的按钮栏，如图9-24所示。

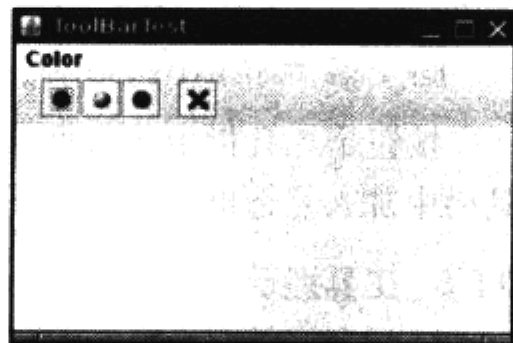


图9-24 工具栏

工具栏的特殊之处在于可以将它随处移动。可以将它拖拽到框架的四个边框上，如图9-25所示。释放鼠标按钮后，工具栏将会停靠在新的位置上，如图9-26所示。

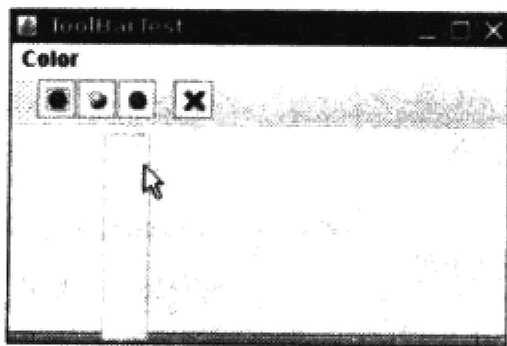


图9-25 拖拽工具栏

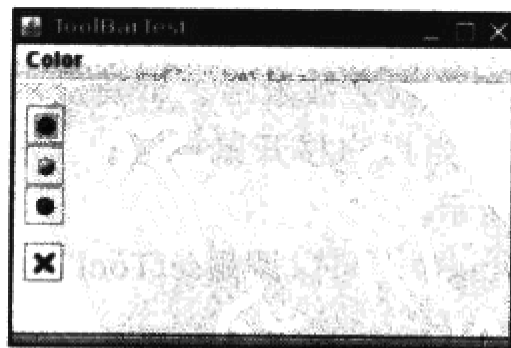


图9-26 将工具栏拖拽到另一边框

注释： 工具栏只有位于采用边框布局或者任何支持North、East、South和West约束布局管理器的容器内才能够被拖拽。

工具栏可以完全脱离框架。这样的工具栏将包含在自己的框架中，如图9-27所示。当关闭包含工具栏的框架时，它会回到原始的框架中。

编写创建工具栏的代码非常容易，并且可以将组件添加到工具栏中：

```
JToolBar bar = new JToolBar();
bar.add(blueButton);
```

JToolBar类还有一个用来添加Action对象的方法，可以用Action对象填充工具栏：

```
bar.add(blueAction);
```

这个动作的小图标将会出现在工具栏中。

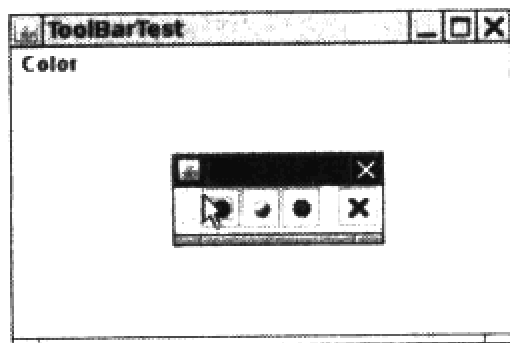


图9-27 脱离的工具栏

可以用分隔符将按钮分组：

```
bar.addSeparator();
```

例如，图9-24中的工具栏有一个分隔符，它位于第三个按钮和第四个按钮之间。

然后，将工具栏添加到框架中：

```
add(bar, BorderLayout.NORTH);
```

当工具栏没有停靠时，可以指定工具栏的标题：

```
bar = new JToolBar(titleString);
```

在默认情况下，工具栏最初为水平的。如果想要将工具栏垂直放置，可以使用下列代码：

```
bar = new JToolBar(SwingConstants.VERTICAL)
```

或者

```
bar = new JToolBar(titleString, SwingConstants.VERTICAL)
```

按钮是工具栏中最常见的组件类型。然而工具栏中的组件并不仅限如此。例如，可以往工具栏中加入复选框。

9.5.8 工具提示

工具栏有一个缺点，这就是用户常常需要猜测按钮上小图标按钮的含义。为了解决这个问题，用户界面设计者发明了工具提示（tooltips）。当光标停留在某个按钮上片刻时，工具提示就会被激活。工具提示文本显示在一个有颜色的矩形里。当用户移开鼠标时，工具提示就会自动地消失。如图9-28所示。

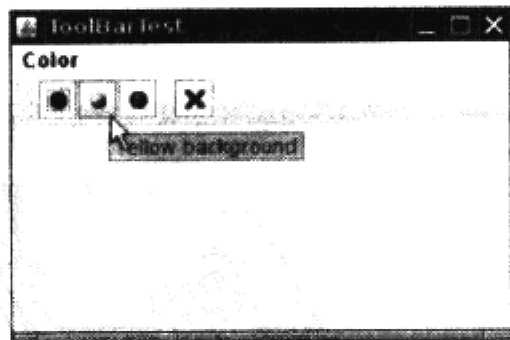


图9-28 工具提示

在Swing中，可以调用setToolText方法将工具提示添加到JComponent上：

```
exitButton.setToolTipText("Exit");
```

还有一种方法是，如果使用Action对象，就可以用SHORT_DESCRIPTION关联工具提示：

```
exitAction.putValue(Action.SHORT_DESCRIPTION, "Exit");
```

例9-9说明了如何将一个Action对象添加到菜单和工具栏中。注意，动作名在菜单中就是菜单项名，而在工具栏中就是简短的说明。

例9-9 ToolBarTest.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * @version 1.13 2007-06-12
 * @author Cay Horstmann
 */
public class ToolBarTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               ToolBarFrame frame = new ToolBarFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * A frame with a toolbar and menu for color changes.
 */
class ToolBarFrame extends JFrame
{
   public ToolBarFrame()
   {
      setTitle("ToolBarTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // add a panel for color change

      panel = new JPanel();
      add(panel, BorderLayout.CENTER);

      // set up actions

      Action blueAction = new ColorAction("Blue", new ImageIcon("blue-ball.gif"), Color.BLUE);
      Action yellowAction = new ColorAction("Yellow", new ImageIcon("yellow-ball.gif"),
            Color.YELLOW);
      Action redAction = new ColorAction("Red", new ImageIcon("red-ball.gif"), Color.RED);

      Action exitAction = new AbstractAction("Exit", new ImageIcon("exit.gif"))
         {
            public void actionPerformed(ActionEvent event)
            {
               System.exit(0);
            }
         };
      exitAction.putValue(Action.SHORT_DESCRIPTION, "Exit");

```

```
        // populate tool bar

        JToolBar bar = new JToolBar();
        bar.add(blueAction);
        bar.add(yellowAction);
        bar.add(redAction);
        bar.addSeparator();
        bar.add(exitAction);
        add(bar, BorderLayout.NORTH);

        // populate menu

        JMenu menu = new JMenu("Color");
        menu.add(yellowAction);
        menu.add(blueAction);
        menu.add(redAction);
        menu.add(exitAction);
        JMenuBar menuBar = new JMenuBar();
        menuBar.add(menu);
        setJMenuBar(menuBar);
     }

     public static final int DEFAULT_WIDTH = 300;
     public static final int DEFAULT_HEIGHT = 200;

     private JPanel panel;

     /**
      * The color action sets the background of the frame to a given color.
      */
     class ColorAction extends AbstractAction
     {
        public ColorAction(String name, Icon icon, Color c)
        {
           putValue(Action.NAME, name);
           putValue(Action.SMALL_ICON, icon);
           putValue(Action.SHORT_DESCRIPTION, name + " background");
           putValue("Color", c);
        }

        public void actionPerformed(ActionEvent event)
        {
           Color c = (Color) getValue("Color");
           panel.setBackground(c);
        }
     }
}
```



javax.swing.JToolBar 1.2

- JToolBar()
- JToolBar(String titleString)
- JToolBar(int orientation)
- JToolBar(String titleString, int orientation)

用给定的标题字符串和方位构造一个工具栏。Orientation可以是SwingConstants.HORIZONTAL（默认）或SwingConstants.VERTICAL。

- `JButton add(Action a)`

  用给定的动作名、图标、简要的说明和动作回调构造一个工具栏中的新按钮。
- `void addSeparator()`

  将一个分隔符添加到工具栏的尾部。

API `javax.swing.JComponent 1.2`

- `void setToolTipText(String text)`

  设置当鼠标停留在组件上时显示在工具提示中的文本。

9.6 复杂的布局管理

迄今为止．在前面的示例应用程序所使用的用户界面组件中，只使用了边框布局、流布局和网格布局。对于复杂的问题而言，只使用这四种布局显然不够。本节将详细地讨论高级布局管理器。

Windows程序员可能会为Java对布局管理器如此兴师动众而感到奇怪。毕竟，在Windows中，布局管理不是一个太大的问题：首先，可以用对话框编辑器将组件拖放到对话柜的表面上。然后，再使用编辑器工具完成组件对齐、均衡间隔、中心定位等工作。如果正在开发的是一个大型项目，可能根本就不必担心组件如何布局，技术娴熟的用户界面设计师会完成所有这些任务。

使用这种方法布局会出现这个的问题：如果组件的大小发生改变，必须手工地更新。为什么组件的大小会发生改变呢？通常有两种可能。第一种可能是为按钮标签和其他对话框文本选择了一种较大的字体。如果在Windows里试验一下就会发现很多应用程序没有解决好这个问题。按钮的尺寸不增大，大字体被紧缩在原来的空间里。当应用程序中的字符串翻译成其他语言时也有可能出现同样的问题，例如，“Cancel”在德语中为“Abbrechen”。如果一个按钮的大小被设计成刚好能够显示字符串“Cancel”，那么德语版显示就会出现问题了，字符串将会被剪掉一部分。

为什么Windows中的按钮不能动态地增大以适应标签呢？这是因为用户界面设计师没有给出应该在哪个方向增大的指令。每个组件拖放或排列之后，对话框编辑器只保存其像素位置和尺寸大小。至于组件为什么以这种方式排列并没有记录下来。

Java布局管理器是一种用于组件布局的好方法。应用布局管理器，布局就可以使用组件间关系的指令来完成布局操作。对于最初的AWT来说，这一点特别重要，这是因为AWT使用的是本地用户界面元素。在Motif、Windows和Macintosh中，按钮和列表框的大小各不相同，而且应用程序或applet不会预先知道它们将在哪个平台上显示。在某种程度上，可变性在Swing中就没有那么重要。如果应用程序强制使用特定的观感，如Metal观感，那么这个程序在所有平台上显示的结果都一样。但是，如果允许应用程序的用户随意地选择观感，则需要依据布局管理器的灵活性调整组件的排列了。

自从Java 1.0以来，AWT就含有*网格组布局*（grid bag layout），这种布局将组件按行和列排列。行和列的大小可以灵活改变，并且组件可以横跨多行多列。这种布局管理器非常灵活，但也非常复杂。仅仅提及“网格组布局”一词就会吓住一些Java程序员。

Swing设计者有一个失败的尝试：为了能够将程序员从使用网格组布局的困难中解脱出来，提出了一种被称为*箱式布局*（box layout）的布局管理器。在BoxLayout类的JDK文档中写到：“采用水平和垂直[sic]的不同组合内嵌多个面板将可以获得与GridBagLayout类似的效果，而且降低了复杂度。”然而，由于每个箱子是独立布局的，所以不能使用箱式布局排列水平和垂直方向都相邻的组件。

Java SE1.4还做了一个尝试：设计一种代替网格组组件的布局——（spring layout）。这种布局使用一个虚构的弹簧将同一个容器中的所有组件连接起来。当容器改变大小时，弹簧可以伸展或收缩，以便调节组件的位置。这听起来似乎感觉枯燥且不可思议，其实也确实如此。弹簧布局很快就会陷入含糊不清的境地。

在2005年，NetBeans开发队伍发明了Matisse技术，这种技术将布局工具与布局管理器结合起来。用户界面设计者可以使用工具将组件拖拽到容器中，并指出组件的排列方式。工具将设计者的意图转换成组布局管理器的可以理解的指令，与手工地编写布局管理的代码相比，这样做要便捷得多。组布局管理器是Java SE 6中的一个新特性。即使没有用NetBeans作为IDE，也应该考虑使用它的GUI建造工具。可以用NetBeans设计GUI，然后再将得到的代码粘贴到所选择的IDE中。

接下来，将讲述网格组布局。这是因为这种布局在早期的Java版本中，使用的最普遍，且也是产生布局代码的最简单方式。下面的策略可以让网格组布局的使用相对简单些。

随后，介绍Matisse工具和组布局管理器。我们需要了解组布局管理器的工作过程，以便能够在用可视化方式将组件放置在某个位置时能够查看Matisse记录的指令是否正确。

最后，将演示如何完全不使用布局管理，手工地放置组件，以及如何编写自己的布局管理器。

9.6.1 网格组布局

网格组布局是所有布局管理器之首。可以将网格组布局看成是没有任何限制的网格布局。在网格组布局中，行和列的尺寸可以改变。可以将相邻的单元合并以适应较大的组件（很多字处理器以及HTML都利用这个功能编辑表格：一旦需要就合并相邻的单元格）。组件不需要填充整个单元格区域，并可以指定它们在单元格内的对齐方式。

请看图9-29中所示的字体选择对话框，其中包含下面的组件：

- 两个用于指定字体外观和大小的组合框
- 两个组合框的标签
- 两个用于选择粗体和斜体的复选框
- 一个用于显示示例字符串的本文区

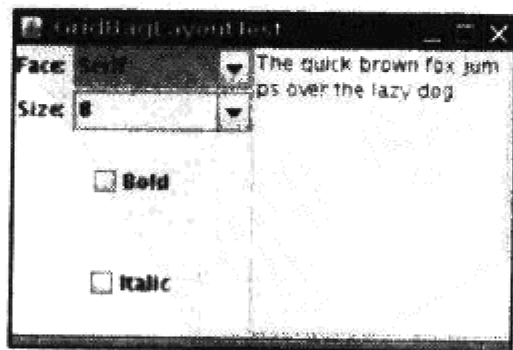


图9-29 字体对话框

现在，将容器分解为网格单元，如图9-29所示（行和列的

尺寸不需要相同)。每个复选框横跨两列,文本区横跨四行。

要想使用网格组管理器进行布局,必须经过下列过程:

1)建立一个GridBagLayout的对象。不需要指定网格的行数和列数。布局管理器会根据后面所给的信息猜测出来。

2)将GridBagLayout对象设置成组件的布局管理器。

3)为每个组件建立一个GridBagConstraints对象。设置GridBagConstraints对象的域以便指出组件在网格组中的布局方案。

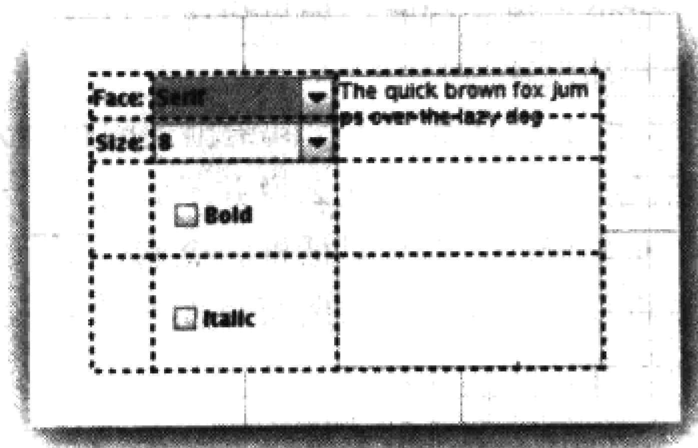


图9-30 设计中使用的对话框网格

4)最后,通过下面的调用添加组件的约束:

```
add(component, constraints);
```

下面给出了相应的代码(稍后将更加详细地介绍各种约束。如果现在不明白约束的作用,不必担心)。

```
GridBagLayout layout = new GridBagLayout();
panel.setLayout(layout);
GridBagConstraints constraints = new GridBagConstraints();
constraints.weightx = 100;
constraints.weighty = 100;
constraints.gridx = 0;
constraints.gridy = 2;
constraints.gridwidth = 2;
constraints.gridheight = 1;
panel.add(component, constraints);
```

知道如何设置GridBagConstraints对象的状态是非常重要的。下面将详细地介绍几个最重要的约束。

1. gridx、gridy、gridwidth和 gridheight 参数

这些约束定义了组件在网格中的位置。gridx和gridy指定了被添加组件左上角的行、列位置。gridwidth和gridheight指定了组件占据的行数和列数。

网格的坐标从0开始。gridx=0和gridy=0代表最左上角。例如,示例程序中,文本区的gridx=2,gridy=0。这是因为这个文本区起始于0行2列(即第3列),girdwidth=1,gridheight=4因为它横跨了4行1列。

2. 增量域

在网格布局中,需要为每个区域设置*增量域*(weightx和weighty)。如果将增量设置为0,则这个区域将永远为初始尺寸。在如图9-29所示的网格布局中,由于将标签的weightx设置为0,所以在窗口缩放时,标签大小始终保持不变。另一方面,如果将所有区域的增量都设置为0,容器就会集聚在为它分配的区域中间,而不是通过拉伸来填充它。

从概念上讲,增量参数属于行和列的属性,而不属于某个单独的单元格。但却需要在单元

格上指定它们，这是因为网格组布局并不暴露行和列。行和列的增量等于每行或每列单元格的增量最大值。因此，如果想让一行或一列的大小保持不变，就需要将这行、这列的所有组件的增量都设置为0。

注意，增量并不实际给出列的相对大小。当容器超过首选大小时，增量表示分配给每个区域的扩展比例值。这么说并不太直观。这里建议将所有的增量设置为100，运行程序，查看一下布局情况。缩放对话框，查看一下行和列是如何调整的。如果发现某行或某列不应该扩大，就将那行或那列中的所有组件的增量设置为0。也可以使用其他的增量值进行修补，但是这么做的意义并不大。

3. fill和anchor参数

如果不希望组件拉伸至整个区域，就需要设置fill约束。它有四个有效值：GridBagConstraints.NONE、GridBagConstraints.HORIZONTAL、GridBagConstraints. VERTICAL和GridBagConstraints.BOTH.。

如果组件没有填充整个区域，可以通过设置anchor域指定其位置。有效值为GridBagConstraints.CENTER(默认值)、GridBagConstraints.NORTH、GridBagConstraints. NORTHEAST和GridBagConstraints.EAST等。

4. 填塞

可以通过设置GridBagLayout的insets域在组件周围增加附加的空白区域。通过设置Insets对象的left、top、right 和bottom指定组件周围的空间量。这被称作*外部填塞*（external padding）。

通过设置ipadx和ipady指定*内部填塞*（internal padding）。这两个值被加到组件的最小宽度和最小高度上。这样可以保证组件不会收缩至最小尺寸之下。

5. 指定gridx, gridy, gridwidth和gridheight参数的另一种方法

AWT文档建议不要将gridx和gridy设置为绝对位置，应该将它们设置为常量GridBagConstraints.RELATIVE。然后，按照标准的顺序，将组件添加到网格组布局中。即第一行从左向右，然后再开始新的一行，以此类推。

还需要通过为gridheight和gridwidth域指定一个适当的值来设置组件横跨的行数和列数。除此之外，如果组件扩展至最后一行或最后一列，则不要给出一个实际的数值，而是用常量GridBagConstraints.REMAINDER替代，这样会告诉布局管理器这个组件是本行上的最后一个组件。

这种方案看起来能起作用，但似乎显得有点笨拙。这是因为这样做会将实际位置信息对布局管理器隐藏起来，而日后又希望它能够重新发现这些信息。

这些事情看起来都很麻烦和复杂，但实际上，下面的策略可以让网格组布局的使用相对简单一些：

1）在纸上画出组件布局草图。

2）找出一种网格，小组件被放置在一个单元格内，大组件将横跨多个单元格。

3）用0，1，2……标识网格的行和列。现在可以读取gridx, gridy, gridwidth和gridheight的值。

4）对于每个组件，需要考虑下列问题：是否需要水平或者垂直填充它所在的单元格？如果不需要，希望如何排列？这些就是fill和anchor参数的设置。

5）将所有的增量设置为100。如果需要某行或某列始终保持默认的大小，就将这行或这列中所有组件的weightx和weighty设置为0。

6）编写代码。仔细地检查GridBagConstraints的设置。错误的约束可能会破坏整个布局。

7）编译、运行。

有些GUI构造者使用工具来可视化地指定约束。如图9-31就是NetBeans中的配置对话框。

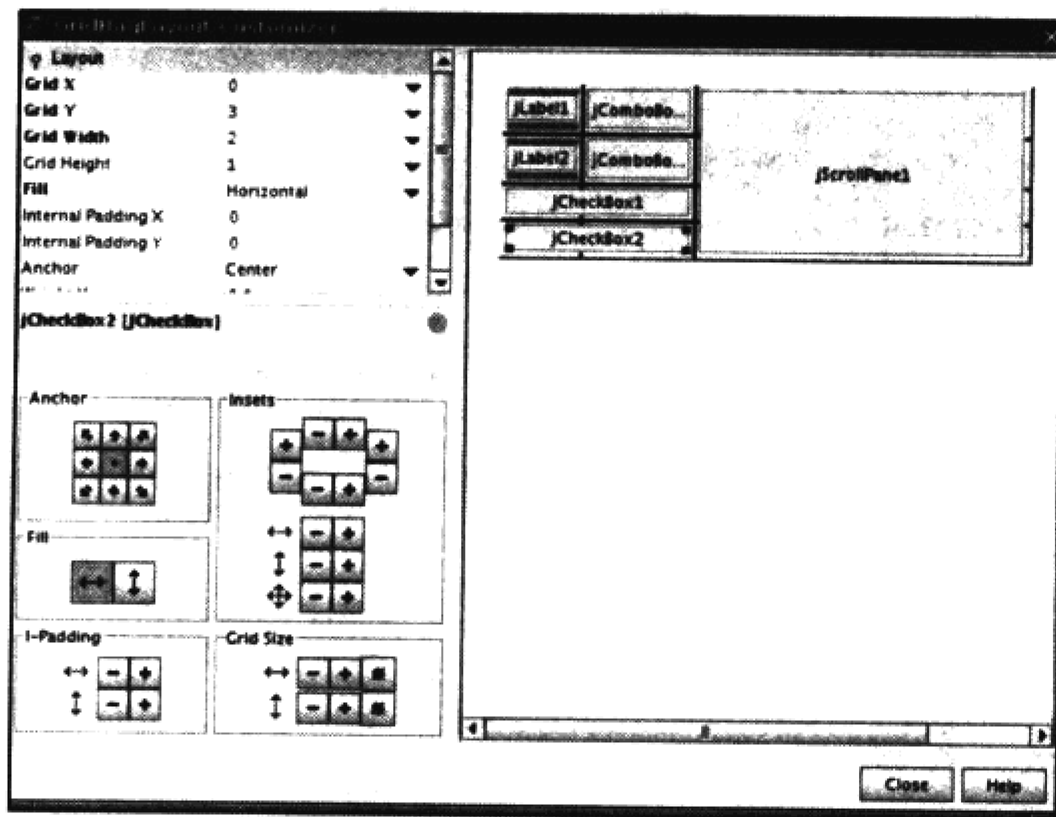


图9-31 在NetBeans中指定网格组的约束

6. 使用帮助类来管理（tame）网格组约束

网格组布局最乏味的工作就是为设置约束编写代码。为此，很多程序员编写帮助函数或者帮助类来满足上面的目的。下面是为字体对话框示例编写的帮助类。这个类有下列特性：

- 名字简短：GBC代替GridBagConstraints。
- 扩展于GridBagConstraints，因此可以使用约束的缩写，如GBC.EAST。
- 当添加组件时，使用GBC对象，如：

```
add(component, new GBC(1, 2));
```

- 有两个构造器可以用来设置最常用的参数：gridx和gridy，或者gridx、gridy、gridwidth和gridheight。

```
add(component, new GBC(1, 2, 1, 4));
```

- 域有很便捷的设置器setter，x/y值对：

```
add(component, new GBC(1, 2).setWeight(100, 100));
```

- Setter方法将返回this，所以可以链接它们：

```
add(component, new GBC(1, 2).setAnchor(GBC.EAST).setWeight(100, 100));
```

• setInsets方法将构造Inset对象。要想获取1个像素的insets，可以调用：

```
add(component, new GBC(1, 2).setAnchor(GBC.EAST).setInsets(1));
```

例9-10显示了字体对话框示例的全部代码。下面是将组件添加到网格组中的代码：

```
add(faceLabel, new GBC(0, 0).setAnchor(GBC.EAST));
add(face, new GBC(1, 0).setFill(GBC.HORIZONTAL).setWeight(100, 0).setInsets(1));
add(sizeLabel, new GBC(0, 1).setAnchor(GBC.EAST));
add(size, new GBC(1, 1).setFill(GBC.HORIZONTAL).setWeight(100, 0).setInsets(1));
add(bold, new GBC(0, 2, 2, 1).setAnchor(GBC.CENTER).setWeight(100, 100));
add(italic, new GBC(0, 3, 2, 1).setAnchor(GBC.CENTER).setWeight(100, 100));
add(sample, new GBC(2, 0, 1, 4).setFill(GBC.BOTH).setWeight(100, 100));
```

一旦理解了网格组约束，就会觉得这些代码十分易于阅读且便于调试。



注释：Sun在http://java.sun.com/docs/books/tutorial/uiswing/layout/gridbag.html的指南中建议：对于所有的组件重用同一个GridBagConstraints对象。我们发现这样做将会使代码难于阅读并易于发生错误。例如，请看http://java.sun.com/docs/books/tutorial/uiswing/events/containerlistener.html的演示。按钮真的被水平拉伸吗？还是程序员忘记了关闭fill约束设定的BOTH？

例9-10 GridBagLayoutTest.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * @version 1.33 2007-06-12
 * @author Cay Horstmann
 */
public class GridBagLayoutTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               FontFrame frame = new FontFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * A frame that uses a grid bag layout to arrange font selection components.
 */
class FontFrame extends JFrame
{
   public FontFrame()
   {
      setTitle("GridBagLayoutTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
```

```
        GridBagLayout layout = new GridBagLayout();
        setLayout(layout);

        ActionListener listener = new FontAction();

        // construct components

        JLabel faceLabel = new JLabel("Face: ");

        face = new JComboBox(new String[] { "Serif", "SansSerif", "Monospaced", "Dialog",
              "DialogInput" });

        face.addActionListener(listener);

        JLabel sizeLabel = new JLabel("Size: ");

        size = new JComboBox(new String[] { "8", "10", "12", "15", "18", "24", "36", "48" });

        size.addActionListener(listener);

        bold = new JCheckBox("Bold");
        bold.addActionListener(listener);

        italic = new JCheckBox("Italic");
        italic.addActionListener(listener);

        sample = new JTextArea();
        sample.setText("The quick brown fox jumps over the lazy dog");
        sample.setEditable(false);
        sample.setLineWrap(true);
        sample.setBorder(BorderFactory.createEtchedBorder());

        // add components to grid, using GBC convenience class

        add(faceLabel, new GBC(0, 0).setAnchor(GBC.EAST));
        add(face, new GBC(1, 0).setFill(GBC.HORIZONTAL).setWeight(100, 0).setInsets(1));
        add(sizeLabel, new GBC(0, 1).setAnchor(GBC.EAST));
        add(size, new GBC(1, 1).setFill(GBC.HORIZONTAL).setWeight(100, 0).setInsets(1));
        add(bold, new GBC(0, 2, 2, 1).setAnchor(GBC.CENTER).setWeight(100, 100));
        add(italic, new GBC(0, 3, 2, 1).setAnchor(GBC.CENTER).setWeight(100, 100));
        add(sample, new GBC(2, 0, 1, 4).setFill(GBC.BOTH).setWeight(100, 100));
    }

    public static final int DEFAULT_WIDTH = 300;
    public static final int DEFAULT_HEIGHT = 200;

    private JComboBox face;
    private JComboBox size;
    private JCheckBox bold;
    private JCheckBox italic;
    private JTextArea sample;

    /**
     * An action listener that changes the font of the sample text.
     */
    private class FontAction implements ActionListener
    {
```

```java
      public void actionPerformed(ActionEvent event)
      {
         String fontFace = (String) face.getSelectedItem();
         int fontStyle = (bold.isSelected() ? Font.BOLD : 0)
               + (italic.isSelected() ? Font.ITALIC : 0);
         int fontSize = Integer.parseInt((String) size.getSelectedItem());
         Font font = new Font(fontFace, fontStyle, fontSize);
         sample.setFont(font);
         sample.repaint();
      }
   }
}
```

例9-11显示了CBC帮助类的代码。

例9-11 GBC.java

```java
import java.awt.*;

/**
 * This class simplifies the use of the GridBagConstraints class.
 * @version 1.01 2004-05-06
 * @author Cay Horstmann
 */
public class GBC extends GridBagConstraints
{
   /**
    * Constructs a GBC with a given gridx and gridy position and all other grid
    * bag constraint values set to the default.
    * @param gridx the gridx position
    * @param gridy the gridy position
    */
   public GBC(int gridx, int gridy)
   {
      this.gridx = gridx;
      this.gridy = gridy;
   }

   /**
    * Constructs a GBC with given gridx, gridy, gridwidth, gridheight and all
    * other grid bag constraint values set to the default.
    * @param gridx the gridx position
    * @param gridy the gridy position
    * @param gridwidth the cell span in x-direction
    * @param gridheight the cell span in y-direction
    */
   public GBC(int gridx, int gridy, int gridwidth, int gridheight)
   {
      this.gridx = gridx;
      this.gridy = gridy;
      this.gridwidth = gridwidth;
      this.gridheight = gridheight;
   }

   /**
    * Sets the anchor.
```

```
    * @param anchor the anchor value
    * @return this object for further modification
    */
   public GBC setAnchor(int anchor)
   {
      this.anchor = anchor;
      return this;
   }

   /**
    * Sets the fill direction.
    * @param fill the fill direction
    * @return this object for further modification
    */
   public GBC setFill(int fill)
   {
      this.fill = fill;
      return this;
   }

   /**
    * Sets the cell weights.
    * @param weightx the cell weight in x-direction
    * @param weighty the cell weight in y-direction
    * @return this object for further modification
    */
   public GBC setWeight(double weightx, double weighty)
   {
      this.weightx = weightx;
      this.weighty = weighty;
      return this;
   }

   /**
    * Sets the insets of this cell.
    * @param distance the spacing to use in all directions
    * @return this object for further modification
    */
   public GBC setInsets(int distance)
   {
      this.insets = new Insets(distance, distance, distance, distance);
      return this;
   }

   /**
    * Sets the insets of this cell.
    * @param top the spacing to use on top
    * @param left the spacing to use to the left
    * @param bottom the spacing to use on the bottom
    * @param right the spacing to use to the right
    * @return this object for further modification
    */
   public GBC setInsets(int top, int left, int bottom, int right)
   {
      this.insets = new Insets(top, left, bottom, right);
      return this;
   }
```

```

   /**
    * Sets the internal padding
    * @param ipadx the internal padding in x-direction
    * @param ipady the internal padding in y-direction
    * @return this object for further modification
    */
   public GBC setIpad(int ipadx, int ipady)
   {
      this.ipadx = ipadx;
      this.ipady = ipady;
      return this;
   }
}
```

API java.awt.GridBagConstraints 1.0

- `int gridx, gridy`

  指定单元格的起始行和列。默认值为0。

- `int gridwidth, gridheight`

  指定单元格的行和列的范围。默认值为1。

- `double weightx, weighty`

  指定单元格扩大时的容量。默认值为0。

- `int anchor`

  表示组件在单元格内的对齐方式。可以选择的绝对位置有：

NORTHWEST	NORTH	NORTHEAST
WEST	CENTER	EAST
SOUTHWEST	SOUTH	SOUTHEAST

  或者各个方向上的相对位置：

FIRST_LINE_START	LINE_START	FIRST_LINE_END
PAGE_START	CENTER	PAGE_END
LAST_LINE_START	LINE_END	LAST_LINE_END

  如果应用程序有可能从右向左，或者自顶至底排列文本，就应该使用后者。默认值为CENTER。

- `int fill`

  指定组件在单元格内的填充行为，取值为NONE, BOTH, HORIZONTAL，或者VERTICAL。默认值为NONE。

- `int ipadx, ipady`

  指定组件周围的“内部”填充。默认值为0。

- `Insets insets`

  指定组件边框周围的“外部”填充。默认为不填充。

- `GridBagConstraints(int gridx, int gridy, int gridwidth, int gridheight, double`

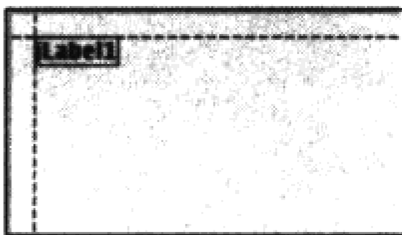
```
weightx, double weighty, int anchor, int fill, Insets insets, int ipadx, int
ipady) 1.2
```

用参数中给定的所有域值构造GridBagConstraints。Sun建议这个构造器只用在自动代码生成器中，因为它会使得代码非常难于阅读。

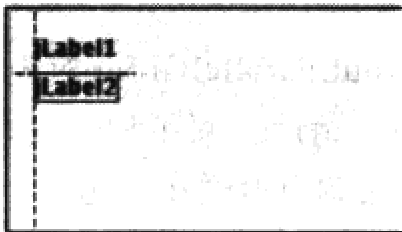
9.6.2 组布局

在讨论GroupLayout类的API之前，先快速地浏览一下NetBeans中的Matisse GUI构造器。这里并没有讲述Matisse全部的使用方法，有关更加详细的信息请参看网站http://netbeans.org/kb/articles/matisse.html。

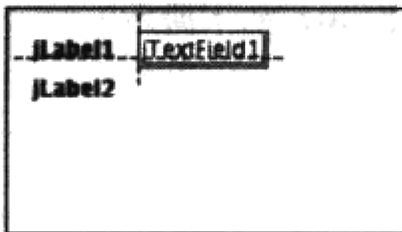
下面是布局如图9-13所示的对话框的工作流程。创建一个新项目，并添加一个新的JFrame表单。拖拽一个标签，直到出现了两条分离于容器边框的引导线：



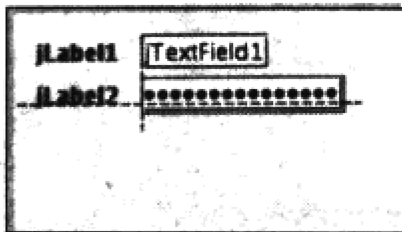
在第一行下面放置另一个标签：



拖拽一个文本域，让文本域的基线与第一个标签的基线对齐。再次注意引导线：



最后，放置一个密码域，它的左侧是标签，上方是文本域。



Matisse将这些操作转换成下列Java代码：

```
layout.setHorizontalGroup(
    layout.createParallelGroup(GroupLayout.Alignment.LEADING)
    .addGroup(layout.createSequentialGroup()
        .addContainerGap()
```

```
            .addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING)
                .addGroup(layout.createSequentialGroup()
                    .addComponent(jLabel1)
                    .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
                    .addComponent(jTextField1))
                .addGroup(layout.createSequentialGroup()
                    .addComponent(jLabel2)
                    .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
                    .addComponent(jPasswordField1)))
            .addContainerGap(222, Short.MAX_VALUE)));
    layout.setVerticalGroup(
        layout.createParallelGroup(GroupLayout.Alignment.LEADING)
        .addGroup(layout.createSequentialGroup()
            .addContainerGap()
            .addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
                .addComponent(jLabel1)
                .addComponent(jTextField1))
            .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
            .addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
                .addComponent(jLabel2)
                .addComponent(jPasswordField1))
            .addContainerGap(244, Short.MAX_VALUE)));
```

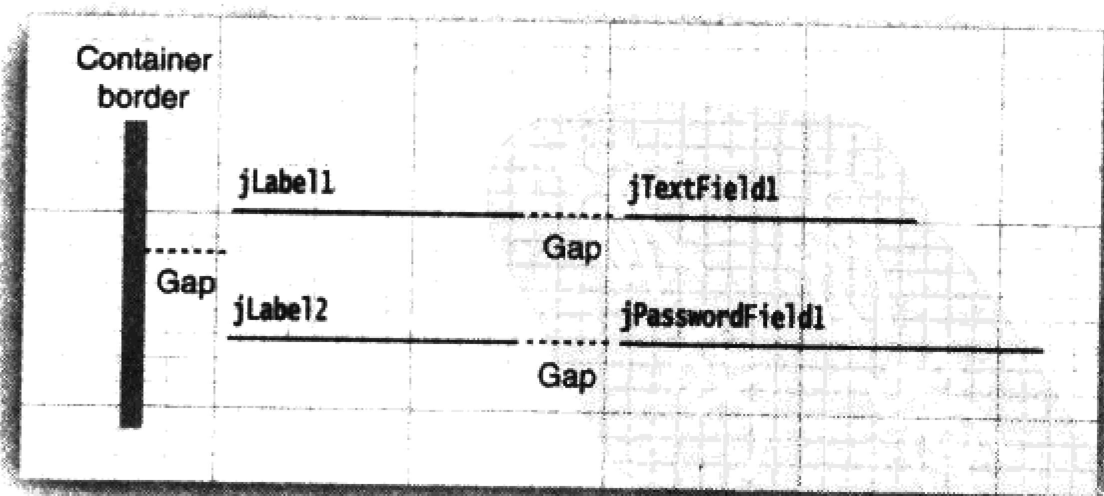
看起来有点让人感觉惊恐，庆幸的是不需要编写代码。阅读一下这段代码会有助于理解布局行为的操作过程，以便能够发现程序中的错误。下面分析一下这段代码的基本结构。在本节后面有关API注解中将更加详细地解释每个类和方法。

可以通过将组件放入GroupLyout.SequentialGroup或者GroupLayout.ParallelGroup对象中将它们组织起来。这些类是GroupLayout.Group的子类。在组中可以包含组件、间距和内嵌的组。由于组类中的各种add方法都返回组对象，因此可以像下面这样将方法调用串联在一起：

```
group.addComponent(...).addPreferredGap(...).addComponent(...);
```

正像从示例代码中所看到的，组布局分别对水平和垂直布局进行计算。

为了能够看到水平计算的效果，假设组件都被拍平了，因此高度为0，如下所示：



有两个平行的组件序列，对应的代码（略有简化）是：

```
.addContainerGap()
.addGroup(layout.createParallelGroup()
    .addGroup(layout.createSequentialGroup()
```

```
        .addComponent(jLabel1)
        .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
        .addComponent(jTextField1))
    .addGroup(layout.createSequentialGroup()
        .addComponent(jLabel2)
        .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
        .addComponent(jPasswordField1)))
```

但是，请稍等，上面这段代码有问题。如果两个标签的长度不一样，那么文本域和密码域就无法对齐。

必须通告Matisse，这里希望将组件对齐。选择这两个域，然后点击鼠标右键，并从菜单中选择Align→Left to Column。同样可以对齐标签。如图9-32所示。

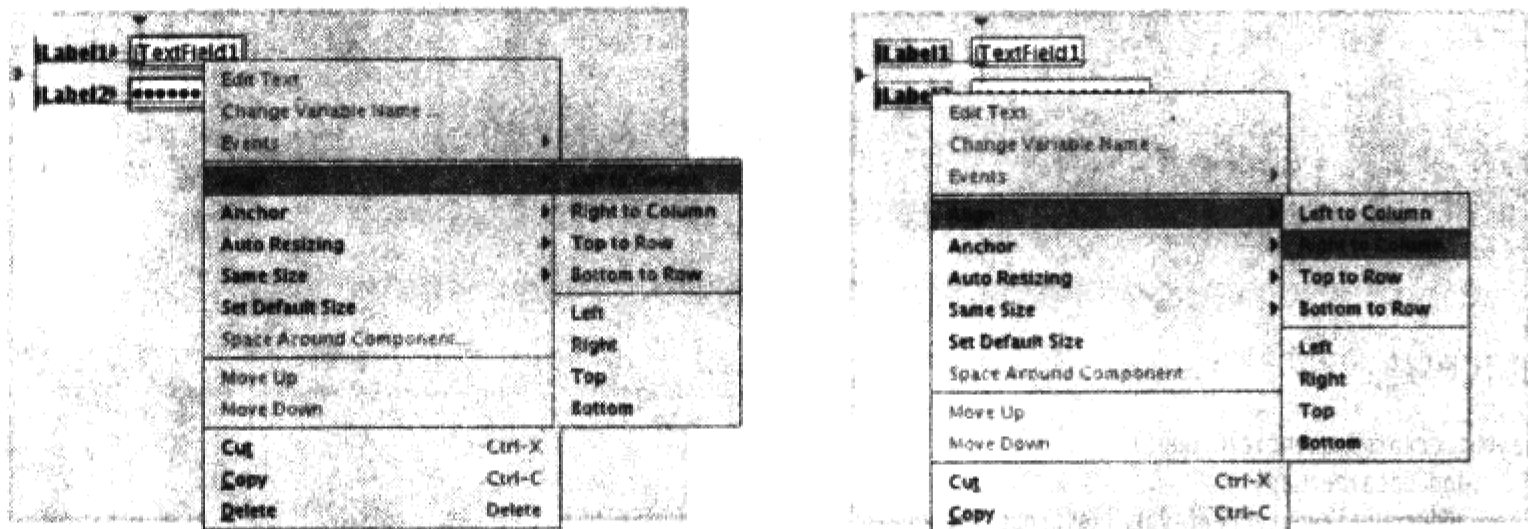
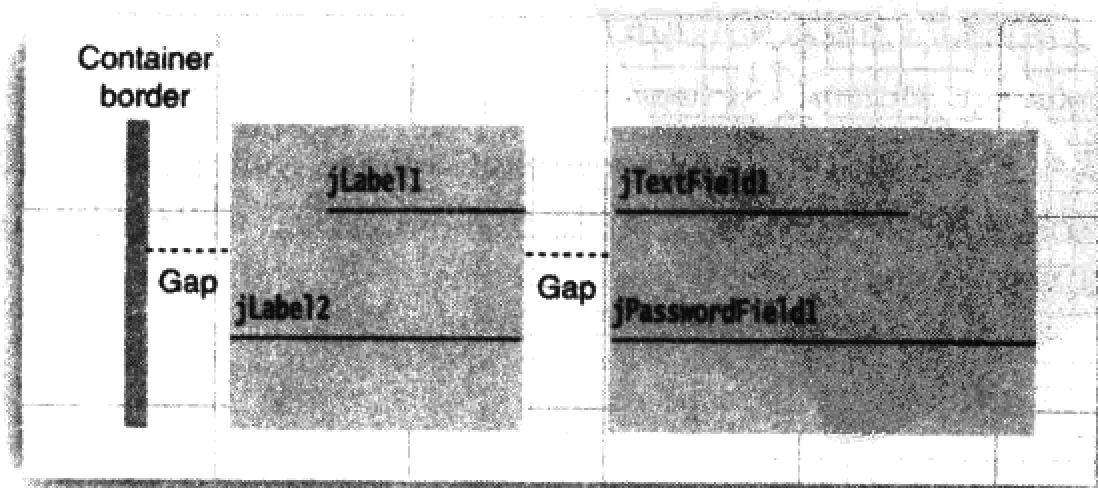


图9-32 在Matisse中对齐标签和文本域

这样做后，戏曲性地改变了布局代码：

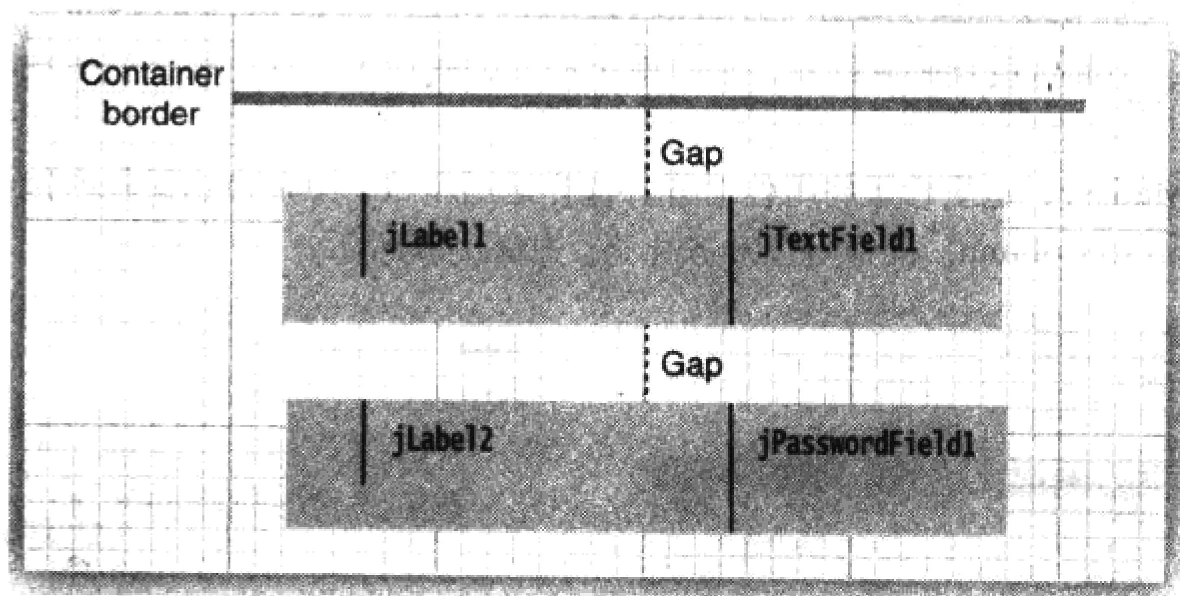
```
.addGroup(layout.createSequentialGroup()
    .addContainerGap()
    .addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING)
        .addComponent(jLabel1, GroupLayout.Alignment.TRAILING)
        .addComponent(jLabel2, GroupLayout.Alignment.TRAILING))
    .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
    .addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING)
        .addComponent(jTextField1)
        .addComponent(jPasswordField1))
```

现在，标签和域分别置于两个平行组中。第一组的对齐方式是TRAILING（这意味着当文本方向是自左向右时，应该右对齐）



简直太奇妙了！Matisse居然可以将设计者的指令转换成嵌套的组。其实，正如Arthur C.Clarke所说：任何高级技术都源于奇特的想法之中。

鉴于完整性的考虑，下面看一下垂直计算。此时，应该将组件看作没有宽度。这里有一个顺序排列的组，其中包含了两个用间距分隔的平行组：



对应的代码是：

```
layout.createSequentialGroup()
    .addContainerGap()
    .addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
        .addComponent(jLabel1)
        .addComponent(jTextField1))
    .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
    .addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
        .addComponent(jLabel2)
        .addComponent(jPasswordField1))
```

从这段代码可以看到：组件依据基线对齐（基线是组件文本对齐的直线）。

注释：在Java较早的版本中，不可能实现基线的精确对齐。在Java SE6的Component类中增加了一个方法getBaseline，以便能够精确地获得含有文本的组件的基线。

可以将一组组件的大小强制为相等。例如，可能想确保文本域和密码域的宽度相等。在Matisse中，选择这两个组件，然后点击鼠标右键，并从菜单中Same Size→Same Width。如图9-33所示。

Matisse将下面这条语句添加到布局代码中：

```
layout.linkSize(SwingConstants.HORIZONTAL, new Component[] {jPasswordField1, jTextField1});
```

在例9-12的代码中显示了如何用GroupLayout替代GridBagLayout来布局前面讲述的字体选择器。这里的代码看起来可能比例9-10要复杂些，但并不必编写。可以使用Matisse进行布局，然后，对少量的代码整理一下即可。

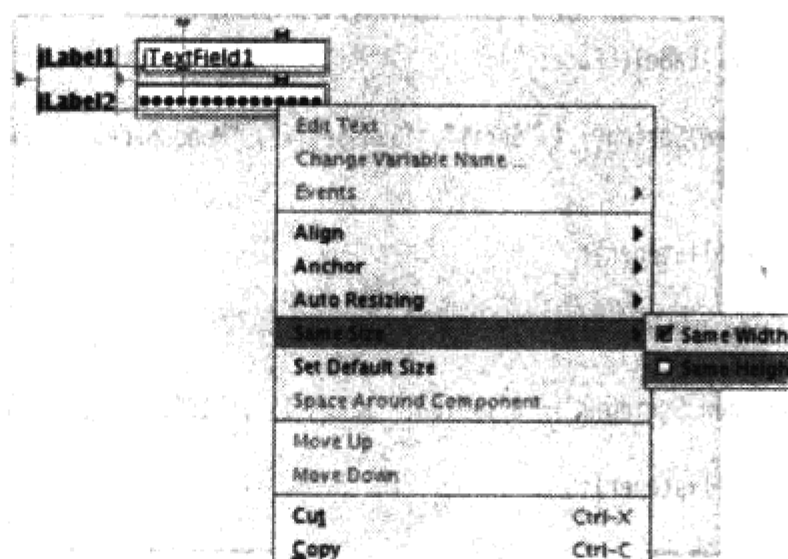


图9-33 强制两个组件的宽度相等

例9-12 GroupLayoutTest.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * @version 1.0 2007-04-27
 * @author Cay Horstmann
 */
public class GroupLayoutTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               FontFrame frame = new FontFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * A frame that uses a group layout to arrange font selection components.
 */
class FontFrame extends JFrame
{
   public FontFrame()
   {
      setTitle("GroupLayoutTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      ActionListener listener = new FontAction();

      // construct components
```

```

      JLabel faceLabel = new JLabel("Face: ");

      face = new JComboBox(new String[] { "Serif", "SansSerif", "Monospaced", "Dialog",
            "DialogInput" });

      face.addActionListener(listener);

      JLabel sizeLabel = new JLabel("Size: ");

      size = new JComboBox(new String[] { "8", "10", "12", "15", "18", "24", "36", "48" });

      size.addActionListener(listener);

      bold = new JCheckBox("Bold");
      bold.addActionListener(listener);

      italic = new JCheckBox("Italic");
      italic.addActionListener(listener);

      sample = new JTextArea();
      sample.setText("The quick brown fox jumps over the lazy dog");
      sample.setEditable(false);
      sample.setLineWrap(true);
      sample.setBorder(BorderFactory.createEtchedBorder());

      pane = new JScrollPane(sample);

      GroupLayout layout = new GroupLayout(getContentPane());
      setLayout(layout);
      layout.setHorizontalGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING)
            .addGroup(
                  layout.createSequentialGroup().addContainerGap().addGroup(
                        layout.createParallelGroup(GroupLayout.Alignment.LEADING).addGroup(
                              GroupLayout.Alignment.TRAILING,
                              layout.createSequentialGroup().addGroup(
                                    layout.createParallelGroup(GroupLayout.Alignment.TRAILING)
                                          .addComponent(faceLabel).addComponent(sizeLabel))
                                    .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
                                    .addGroup(
                                          layout.createParallelGroup(
                                                GroupLayout.Alignment.LEADING, false)
                                                .addComponent(size).addComponent(face)))
                              .addComponent(italic).addComponent(bold)).addPreferredGap(
                        LayoutStyle.ComponentPlacement.RELATED).addComponent(pane)
                        .addContainerGap()));

      layout.linkSize(SwingConstants.HORIZONTAL, new java.awt.Component[] { face, size });

      layout.setVerticalGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING)
            .addGroup(
                  layout.createSequentialGroup().addContainerGap().addGroup(
                        layout.createParallelGroup(GroupLayout.Alignment.LEADING).addComponent(
                              pane, GroupLayout.Alignment.TRAILING).addGroup(
                              layout.createSequentialGroup().addGroup(
                                    layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
                                          .addComponent(face).addComponent(faceLabel))
```

```
                                    .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
                                    .addGroup(
                                        layout.createParallelGroup(
                                           GroupLayout.Alignment.BASELINE).addComponent(size)
                                              .addComponent(sizeLabel)).addPreferredGap(
                                       LayoutStyle.ComponentPlacement.RELATED).addComponent(
                                        italic, GroupLayout.DEFAULT_SIZE,
                                        GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
                                    .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
                                    .addComponent(bold, GroupLayout.DEFAULT_SIZE,
                                        GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)))
                         .addContainerGap()));
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;

   private JComboBox face;
   private JComboBox size;
   private JCheckBox bold;
   private JCheckBox italic;
   private JScrollPane pane;
   private JTextArea sample;

   /**
    * An action listener that changes the font of the sample text.
    */
   private class FontAction implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         String fontFace = (String) face.getSelectedItem();
         int fontStyle = (bold.isSelected() ? Font.BOLD : 0)
               + (italic.isSelected() ? Font.ITALIC : 0);
         int fontSize = Integer.parseInt((String) size.getSelectedItem());
         Font font = new Font(fontFace, fontStyle, fontSize);
         sample.setFont(font);
         sample.repaint();
      }
   }
}
```



javax.swing.GroupLayout 6

- `GroupLayout(Container host)`

  构造一个GroupLayout对象，用于布局host容器中的组件（注意：host容器仍然需要调用setLayout）。

- `void setHorizontalGroup(GroupLayout.Group g)`
- `void setVerticalGroup(GroupLayout.Group g)`

  设置用于控制水平或垂直容器的组。

- `void linkSize(Component... components)`
- `void linkSize(int axis, Component... component)`

强制给定的几个组件具有相同的尺寸，或者在指定的坐标轴上有相同的尺寸（Swing Constants.HORIZONTAL或者SwingContants.VERTICAL）。

- `GroupLayout.SequentialGroup createSequentialGroup()`

  创建一个组，用于平行地布局子组件。

- `GroupLayout.ParallelGroup createParallelGroup()`
- `GroupLayout.ParallelGroup createParallelGroup(GroupLayout.Alignment align)`
- `GroupLayout.ParallelGroup createParallelGroup(GroupLayout.Alignment align,boolean resizable)`

  创建一个组，用于顺序地布局子组件。

  参数：align　　BASELINE、LEADING（默认值）、TRAILING或CENTER

  　　　resizable　如果组可以调整大小，这个值为true（默认值）；如果首选的尺寸是最小尺寸或最大尺寸，这个值为false

- `boolean getHonorsvisibility()`
- `void setHonorsvisibility(boolean b)`

  获取或设置honorsVisibility特性。当这个值为true（默认值）时，不可见的组件不参与布局。当这个值为false时，好像可见的组件一样，这些组件也参与布局。这个特性主要用于想要临时隐藏一些组件，而又不希望改变布局的情况。

- `boolean getAutoCreateCaps()`
- `void setAutoCreateCaps(boolean b)`
- `boolean getAutoCreateContainerCaps()`
- `void setAutoCreateContainerCaps(boolean b)`

  获取或设置autoCreateCaps和autoCreateContainerCaps特性。当这个值为true时，将自动地在组件或容器边框之间添加空隙。默认值是false。在手工地构造GroupLayout时，true值很有用。

API javax.swing.GroupLayout.Group

- `GroupLayout.Group addComponent(Component c)`
- `GroupLayout.Group addComponent(Component c, int minimumSize, int preferredSize, int maximumSize)`

  添加一个组件至本组中。尺寸参数可以是一个实际值（非负值），或者是一个特定的常量GroupLayout.DEFAULT_SIZE或GroupLayout.PREFERRED_SIZE。当使用DEFAULT_SIZE，将调用组件的getMinimumSize、getPreferredSize或getMaximumSize。当使用PREFERRED_SIZE时，将调用getPreferredSize方法。

- `GroupLayout.Group addCap(int size)`
- `GroupLayout.Group addCap(int minimumSize, int preferredSize, int maximumSize)`

  添加一个固定的或可调节的间隙。

- `GroupLayout.Group addGroup(GroupLayout.Group g)`

将一个给定的组添加到本组中。

API `javax.swing.GroupLayout.ParallelGroup`

- `GroupLayout.ParallelGroup addComponent(Component c, GroupLayout.Alignment align)`
- `GroupLayout.ParallelGroup addComponent(Component c, GroupLayout.Alignment align, int minimumSize, int preferredSize, int maximumSize)`
- `GroupLayout.ParallelGroup addGroup(GroupLayout.Group g, GroupLayout.Alignment align)`

  利用给定的对齐方式（BASELINE、LEADING、TRAILINC或CENTER）添加一个组件或组至本组中。

API `javax.swing.GroupLayout.SequentialGroup`

- `GroupLayout.SequentialGroup addContainerCap()`
- `GroupLayout.SequentialGroup addContainerCap(int preferredSize, int maximumSize)`

  为分隔组件和容器的边缘添加一个间隙。
- `GroupLayout.SequentialGroup addPreferredCap(LayoutStyle.ComponentPlacement type)`

  为分隔组件添加一个间隙。间隙的类型是LayoutStyle.ComponentPlacement.RELATED或LayoutStyle.ComponentPlacement。

9.6.3 不使用布局管理器

有时候用户可能不想使用任何布局管理器，而只想把组件放在一个固定的位置上（通常称为*绝对定位*）。这对于与平台无关的应用程序来说并不是一个好主意，但可用来快速地构造原型。

下面是将一个组件定位到某个绝对定位的步骤：

1）将布局管理器设置为null。

2）将组件添加到容器中。

3）指定想要放置的位置和大小。

```
frame.setLayout(null);
JButton ok = new JButton("Ok");
frame.add(ok);
ok.setBounds(10, 10, 30, 15);
```

API `java.awt.Component 1.0`

- `void setBounds(int x, int y, int width, int height)`

  移动并调节组件的尺寸

  参数：x, y　　　　组件新的左上角位置

width, height　　组件新的尺寸

9.6.4 定制布局管理器

原则上，可以通过自己设计LayoutManager类来实现特殊的布局方式。例如，可以将容器中的组件排列成一个圆形。要想达到这个效果，通常很消耗间和精力，但是如图9-34所示，结果可能非常好看。

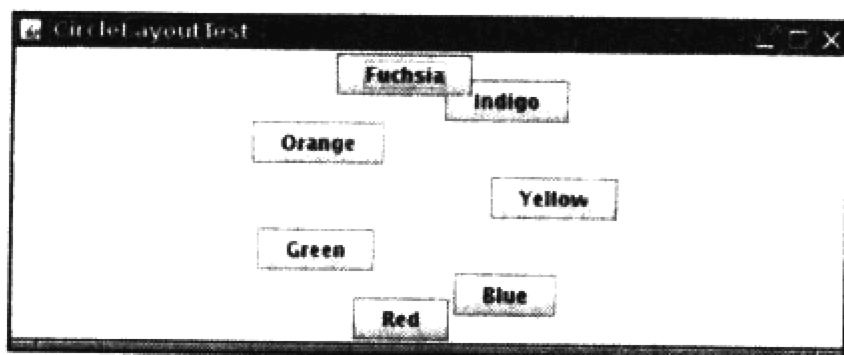


图9-34　圆形布局

如果一定需要定制布局管理器，可以采用下面讲述的定制布局管理器的方法。定制布局管理器必须实现LayoutManager接口，并且需要定义下面5个方法：

```
void addLayoutComponent(String s, Component c);
void removeLayoutComponent(Component c);
Dimension preferredLayoutSize(Container parent);
Dimension minimumLayoutSize(Container parent);
void layoutContainer(Container parent);
```

在添加或删除一个组件时会调用前面两个方法。如果不需要保存组件的任何附加信息，那么可以将这两个方法置为空。接下来的两个方法计算组件的最小布局和首选布局所需要的空间。两者通常相等。第5个方法真正地实施操作，它调用所有组件的setBounds方法。

注释：AWT还有第二个接口LayoutManager2，其中包含10个需要实现的方法，而不是5个。这个接口的主要特点是允许用户使用带有约束的add方法。例如，BorderLayout和GridBagLayout都实现了LayoutManager2接口。

例9-13简单实现了CircleLayout管理器，在父组件中沿着圆形排列组件。这个管理器很有趣，但是没有什么实际的应用价值。

例9-13　CircleLayoutTest.java

```
import java.awt.*;

import javax.swing.*;

/**
 * @version 1.32 2007-06-12
 * @author Cay Horstmann
 */
public class CircleLayoutTest
{
```

```
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               CircleLayoutFrame frame = new CircleLayoutFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * A frame that shows buttons arranged along a circle.
 */
class CircleLayoutFrame extends JFrame
{
   public CircleLayoutFrame()
   {
      setTitle("CircleLayoutTest");

      setLayout(new CircleLayout());
      add(new JButton("Yellow"));
      add(new JButton("Blue"));
      add(new JButton("Red"));
      add(new JButton("Green"));
      add(new JButton("Orange"));
      add(new JButton("Fuchsia"));
      add(new JButton("Indigo"));
      pack();
   }
}

/**
 * A layout manager that lays out components along a circle.
 */
class CircleLayout implements LayoutManager
{
   public void addLayoutComponent(String name, Component comp)
   {
   }

   public void removeLayoutComponent(Component comp)
   {
   }

   public void setSizes(Container parent)
   {
      if (sizesSet) return;
      int n = parent.getComponentCount();

      preferredWidth = 0;
      preferredHeight = 0;
      minWidth = 0;
      minHeight = 0;
```

```
      maxComponentWidth = 0;
      maxComponentHeight = 0;

      // compute the maximum component widths and heights
      // and set the preferred size to the sum of the component sizes.
      for (int i = 0; i < n; i++)
      {
         Component c = parent.getComponent(i);
         if (c.isVisible())
         {
            Dimension d = c.getPreferredSize();
            maxComponentWidth = Math.max(maxComponentWidth, d.width);
            maxComponentHeight = Math.max(maxComponentHeight, d.height);
            preferredWidth += d.width;
            preferredHeight += d.height;
         }
      }
      minWidth = preferredWidth / 2;
      minHeight = preferredHeight / 2;
      sizesSet = true;
   }

   public Dimension preferredLayoutSize(Container parent)
   {
      setSizes(parent);
      Insets insets = parent.getInsets();
      int width = preferredWidth + insets.left + insets.right;
      int height = preferredHeight + insets.top + insets.bottom;
      return new Dimension(width, height);
   }

   public Dimension minimumLayoutSize(Container parent)
   {
      setSizes(parent);
      Insets insets = parent.getInsets();
      int width = minWidth + insets.left + insets.right;
      int height = minHeight + insets.top + insets.bottom;
      return new Dimension(width, height);
   }

   public void layoutContainer(Container parent)
   {
      setSizes(parent);

      // compute center of the circle

      Insets insets = parent.getInsets();
      int containerWidth = parent.getSize().width - insets.left - insets.right;
      int containerHeight = parent.getSize().height - insets.top - insets.bottom;

      int xcenter = insets.left + containerWidth / 2;
      int ycenter = insets.top + containerHeight / 2;

      // compute radius of the circle

      int xradius = (containerWidth - maxComponentWidth) / 2;
      int yradius = (containerHeight - maxComponentHeight) / 2;
```

```
      int radius = Math.min(xradius, yradius);

      // lay out components along the circle

      int n = parent.getComponentCount();
      for (int i = 0; i < n; i++)
      {
         Component c = parent.getComponent(i);
         if (c.isVisible())
         {
            double angle = 2 * Math.PI * i / n;

            // center point of component
            int x = xcenter + (int) (Math.cos(angle) * radius);
            int y = ycenter + (int) (Math.sin(angle) * radius);

            // move component so that its center is (x, y)
            // and its size is its preferred size
            Dimension d = c.getPreferredSize();
            c.setBounds(x - d.width / 2, y - d.height / 2, d.width, d.height);
         }
      }
   }

   private int minWidth = 0;
   private int minHeight = 0;
   private int preferredWidth = 0;
   private int preferredHeight = 0;
   private boolean sizesSet = false;
   private int maxComponentWidth = 0;
   private int maxComponentHeight = 0;
}
```

API java.awt.LayoutManager 1.0

- `void addLayoutComponent(String name, Component comp)`

  将组件添加到布局中。

  参数：name　　组件位置的标识符

  　　　comp　　被添加的组件

- `void removeLayoutComponent(Component comp)`

  从本布局中删除一个组件。

- `Dimension preferredLayoutSize(Container cont)`

  返回本布局下的容器的首选尺寸。

- `Dimension minimumLayoutSize(Container cont)`

  返回本布局中下容器的最小尺寸。

- `void layoutContainer(Container cont)`

  布局容器内的组件。

9.6.5 遍历顺序

当把很多组件添加到窗口中时，需要考虑遍历顺序（traversal order）的问题。窗口被初次显示时，遍历序列的第一个组件会有键盘焦点。每次用户按下TAB键，下一个组件就会获得焦点（回忆一下，具有键盘焦点的组件可以用键盘进行操作。例如，如果按钮具有焦点，按下空格键就相当于“点击”它）。我们可能并不习惯使用TAB键遍历一组控件，但是也有很多用户喜欢这样做。这些人有可能厌恶鼠标，也有可能由于残疾而无法使用鼠标，或者采用语言方式进行交互，所以应该知道Swing是如何设置遍历顺序的。

遍历顺序很直观，它的顺序是从左至右，从上至下。例如，在字体对话框例子中，组件按照下面顺序进行遍历（如图9-35）：

①外观组合框

②示例文本区（按下CTRL+TAB键移动到下一个文本域，TAB字符被认为是文本输入）。

③尺寸组合框

④加粗复选框

⑤斜体复选框

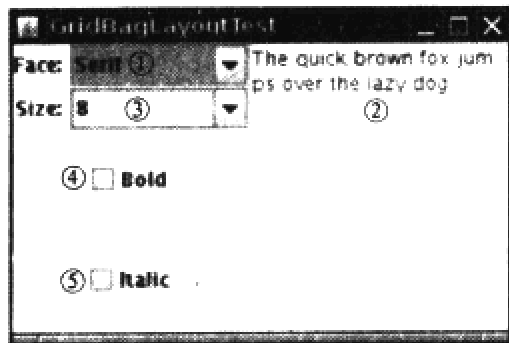


图9-35 地理位置遍历顺序

> **注释**：在早期的AWT中，遍历顺序是由组件插入到容器中的顺序决定。在Swing中，遍历顺序仅取决于组件的布局，而不是插入顺序。

如果容器还包含其他的容器，情况就更加复杂了。当焦点给予另外一个容器时，那个容器左上角的组件就会自动地获得焦点，然后再遍历那个容器中的所有组件。最后，将焦点移交给紧跟着那个容器的组件上。

利用这一点，可以将相关元素组织在一起并放置在一个容器中。例如，放置在一个面板中。

> **注释**：在Java SE 1.4中，调用
>
> ```
> component.setFocusable(false);
> ```
>
> 可以从焦点遍历中删除一个组件。在早期的版本中，必须覆盖isFocusTraversable方法，但是这个方法现在已经不用了。

总之，在Java SE 1.4中有两个标准的遍历策略：

- 纯的AWT应用程序使用DefaultFocusTraversalPolicy。如果组件是可见的，可显示的、激活的、可聚焦的，并且它们的同质对等体也是可聚焦的，那么这些组件就包含在焦点遍历中。组件按照它们插入到容器的先后次序进行遍历。
- Swing应用程序使用LayoutFocusTraversalPolicy。如果组件是可见的，可显示的、激活的以及可聚焦的，那么这些组件就包含在焦点遍历中。组件按照它们的位置顺序进行遍历：从左至右，从上至下。不过，容器将会引入一个新的“循环”，其中的所有组件将会在这个容器的后继组件获得焦点之前被遍历。

> **注释**：“循环”这个概念有点令人迷惑不解。在遍历到子容器的最后一个元素之后，焦点并不回到第一个元素上，而是跳到容器的后继组件上。API支持真正的循环，包括通过去

键在一个循环层次中上下移动。然而，标准的遍历政策不使用层次循环，而是将循环层次变为线性（深度优先）遍历。

注释：在Java SE 1.3中，可以调用JComponent类中的setNextFocusableComponent方法来改变默认的遍历顺序。现在已经不再使用这个方法了。要想改变遍历的顺序，可以将相关的组件放置在一个面板中以形成循环。如果这样不奏效，就需要安装一个比较器，对组件重新进行排序，或者完全替代遍历策略。然而，似乎哪个方法都没有解决根本问题——请参看Sun API文档。

9.7 对话框

到目前为止，所有的用户界面组件都显示在应用程序创建的框架窗口中。这对于编写运行在Web浏览器中的applets来说是十分常见的情况。但是，如果编写应用程序，通常就需要弹出独立的对话框来显示信息或者获取用户信息。

与大多数的窗口系统一样，AWT也分为模式对话框和无模式对话框。所谓模式对话框是指在结束对它的处理之前，不允许用户与应用程序的其余窗口进行交互。模式对话框主要用于在程序继续运行之前获取用户提供的信息。例如，当用户想要读取文件时，就会弹出一个模式对话框。用户必须给定一个文件名，然后程序才能够开始读操作。只有用户关闭（模式）对话框之后，应用程序才能够继续执行。

所谓无模式对话框是指允许用户同时在对话框和应用程序的其他窗口中输入信息。使用无模式对话框的最好例子就是工具栏。工具栏可以停靠在任何地方，并且用户可以在需要的时候，同时与应用程序窗口和工具栏进行交互。

本节从最简单的对话框开始——一个简单信息的模式对话框。Swing有一个很容易使用的类JOptionPane，它可以弹出一个简单的对话框，而不必编写任何对话框的相关代码。随后，将看到如何通过实现自己的对话框窗口来编写一个复杂的对话框。最后，介绍在应用程序与对话框之间如何传递数据。

本节用两个标准的对话框结束：文件对话框和颜色对话框。文件对话框比较复杂，为此需要熟悉Swing中的JFileChooser——自己编写文件对话框是一项颇有挑战性的任务。JColorChooser对话框可用来让用户选取颜色。

9.7.1 选项对话框

Swing有一套简单的对话框，用于获取用户的一些简单信息。JOptionPane有4个用于显示这些对话框的静态方法：

showMessageDialog：	显示一条消息并等待用户点击OK
showConfirmDialog：	显示一条消息并等待用户确认（与OK/Cancel类似）
showOptionDialog：	显示一条消息并获得用户在一组选项中的选择
showInputDialog：	显示一条消息并获得用户输入的一行文本

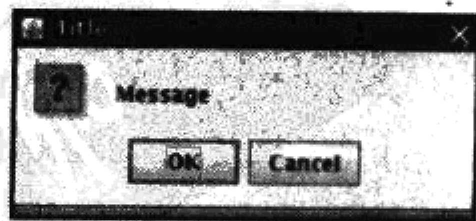


图9-36 选项对话框

图9-36显示了一个典型的对话框。可以看到，对话框有下列组件：

- 一个图标
- 一条消息
- 一个或多个按钮

输入对话框有一个用于接收用户输入的额外组件。它既可能是用于输入任何字符串的文本域，也可能是允许用户从中选择的组合框。

这些对话框的确切布局和为标准消息类型选择的图标都取决于具体的观感。

左侧的图标将由下面5种消息类型决定：

```
ERROR_MESSAGE
INFORMATION_MESSAGE
WARNING_MESSAGE
QUESTION_MESSAGE
PLAIN_MESSAGE
```

PLAIN_MESSAGE类型没有图标。每个对话框类型都有一个方法，可以用来提供自己的图标，以替代原来的图标。

可以为每个对话框类型指定一条消息。这里的消息既可以是字符串、图标、用户界面组件，也可以是其他类型的对象。下面是显示消息对象的基本方式：

String：	绘制字符串
Icon：	显示图标
Component：	显示组件
Object[]：	显示数组中的所有对象，依次叠加
any other object：	调用toString方法来显示结果字符串

可以运行例9-14程序，查看一下这些选项。

当然，提供字符串消息是最常见的情况，而提供一个Component会带来更大的灵活性。这是因为通过调用paintComponent方法可以绘制自己想要的任何内容。

位于底部的按钮取决于对话框类型和选项类型。当调用showMessageDialog 和showInputDialog时，只能看到一组标准按钮（分别是OK/Cancel）。当调用showConfirmDialog时，可以选择下面四种选项类型之一：

```
DEFAULT_OPTION
YES_NO_OPTION
YES_NO_CANCEL_OPTION
OK_CANCEL_OPTION
```

使用showOptionDialog 可以指定任意的选项。这里需要为选项提供一个对象数组。每个数组元素可以是下列类型之一：

String：	使用字符串标签创建一个按钮
Icon：	使用图标创建一个按钮
Component：	显示这个组件

其他类型的对象：使用toString方法，然后用结果字符串作为标签创建按钮

下面是这些方法的返回值：

showMessageDialog	无
showConfirmDialog	表示被选项的一个整数

showOptionDialog	表示被选项的一个整数
showInputDialog	用户选择或输入的字符串

showConfirmDialog和showOptionDialog返回一个整数用来表示用户选择了哪个按钮。对于选项对话框来说，这个值就是被选的选项的索引值或者是CLOSED_OPTION（此时用户没有选择可选项，而是关闭了对话框）。对于确认对话框，返回值可以是下列值之一：

```
OK_OPTION
CANCEL_OPTION
YES_OPTION
NO_OPTION
CLOSED_OPTION
```

这些选项似乎令人感到迷惑不解，实际上非常简单：

1）选择对话框的类型（消息、确认、选项或者输入）。

2）选择图标（错误、信息、警告、问题、无或者自定义）。

3）选择消息（字符串、图表、自定义组件或者它们的集合）。

4）对于确认对话框，选择选项类型（默认、Yes/No、Yes/No/Cancel或者Ok/Cancel）。

5）对于选项对话框，选择选项（字符串、图表或者自定义组件）和默认选项。

6）对于输入对话框，选择文本框或者组合框。

7）调用JOptionPane API中的相应方法。

例如，假设需要显示一个图9-36的对话框。这个对话框显示了一条消息，并请求用户确认或者取消。这是一个确认对话框。图标是警告图标，消息是字符串，选项类型是OK_CANCEL_OPTION。调用如下：

```
int selection = JOptionPane.showConfirmDialog(parent,
   "Message", "Title",
   JOptionPane.OK_CANCEL_OPTION,
   JOptionPane.QUESTION_MESSAGE);
if (selection == JOptionPane.OK_OPTION) . . .
```

! **提示**：*消息字符串中可以包含换行符（'\n'）。这样就可以让字符串显示在多行。*

例9-14中的程序可以让用户进行选择，如图9-37所示，然后显示结果对话框。

例9-14 OptionDialogTest.java

```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import java.util.*;
import javax.swing.*;

/**
 * @version 1.33 2007-04-28
 * @author Cay Horstmann
 */
public class OptionDialogTest
{
   public static void main(String[] args)
```

```
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               OptionDialogFrame frame = new OptionDialogFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * A panel with radio buttons inside a titled border.
 */
class ButtonPanel extends JPanel
{
   /**
    * Constructs a button panel.
    * @param title the title shown in the border
    * @param options an array of radio button labels
    */
   public ButtonPanel(String title, String... options)
   {
      setBorder(BorderFactory.createTitledBorder(BorderFactory.createEtchedBorder(), title));
      setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));
      group = new ButtonGroup();

      // make one radio button for each option
      for (String option : options)
      {
         JRadioButton b = new JRadioButton(option);
         b.setActionCommand(option);
         add(b);
         group.add(b);
         b.setSelected(option == options[0]);
      }
   }

   /**
    * Gets the currently selected option.
    * @return the label of the currently selected radio button.
    */
   public String getSelection()
   {
      return group.getSelection().getActionCommand();
   }

   private ButtonGroup group;
}

/**
 * A frame that contains settings for selecting various option dialogs.
 */
class OptionDialogFrame extends JFrame
{
```

```
public OptionDialogFrame()
{
   setTitle("OptionDialogTest");
   setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

   JPanel gridPanel = new JPanel();
   gridPanel.setLayout(new GridLayout(2, 3));

   typePanel = new ButtonPanel("Type", "Message", "Confirm", "Option", "Input");
   messageTypePanel = new ButtonPanel("Message Type", "ERROR_MESSAGE", "INFORMATION_MESSAGE",
         "WARNING_MESSAGE", "QUESTION_MESSAGE", "PLAIN_MESSAGE");
   messagePanel = new ButtonPanel("Message", "String", "Icon", "Component", "Other", "Object[]");
   optionTypePanel = new ButtonPanel("Confirm", "DEFAULT_OPTION", "YES_NO_OPTION",
         "YES_NO_CANCEL_OPTION", "OK_CANCEL_OPTION");
   optionsPanel = new ButtonPanel("Option", "String[]", "Icon[]", "Object[]");
   inputPanel = new ButtonPanel("Input", "Text field", "Combo box");

   gridPanel.add(typePanel);
   gridPanel.add(messageTypePanel);
   gridPanel.add(messagePanel);
   gridPanel.add(optionTypePanel);
   gridPanel.add(optionsPanel);
   gridPanel.add(inputPanel);

   // add a panel with a Show button

   JPanel showPanel = new JPanel();
   JButton showButton = new JButton("Show");
   showButton.addActionListener(new ShowAction());
   showPanel.add(showButton);

   add(gridPanel, BorderLayout.CENTER);
   add(showPanel, BorderLayout.SOUTH);
}

/**
 * Gets the currently selected message.
 * @return a string, icon, component or object array, depending on the Message panel selection
 */
public Object getMessage()
{
   String s = messagePanel.getSelection();
   if (s.equals("String")) return messageString;
   else if (s.equals("Icon")) return messageIcon;
   else if (s.equals("Component")) return messageComponent;
   else if (s.equals("Object[]")) return new Object[] { messageString, messageIcon,
         messageComponent, messageObject };
   else if (s.equals("Other")) return messageObject;
   else return null;
}

/**
 * Gets the currently selected options.
 * @return an array of strings, icons or objects, depending on the Option panel selection
 */
public Object[] getOptions()
{
```

```
      String s = optionsPanel.getSelection();
      if (s.equals("String[]")) return new String[] { "Yellow", "Blue", "Red" };
      else if (s.equals("Icon[]")) return new Icon[] { new ImageIcon("yellow-ball.gif"),
            new ImageIcon("blue-ball.gif"), new ImageIcon("red-ball.gif") };
      else if (s.equals("Object[]")) return new Object[] { messageString, messageIcon,
            messageComponent, messageObject };
      else return null;
   }

   /**
    * Gets the selected message or option type
    * @param panel the Message Type or Confirm panel
    * @return the selected XXX_MESSAGE or XXX_OPTION constant from the JOptionPane class
    */
   public int getType(ButtonPanel panel)
   {
      String s = panel.getSelection();
      try
      {
         return JOptionPane.class.getField(s).getInt(null);
      }
      catch (Exception e)
      {
         return -1;
      }
   }

   /**
    * The action listener for the Show button shows a Confirm, Input, Message or Option dialog
    * depending on the Type panel selection.
    */
   private class ShowAction implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         if (typePanel.getSelection().equals("Confirm")) JOptionPane.showConfirmDialog(
               OptionDialogFrame.this, getMessage(), "Title", getType(optionTypePanel),
               getType(messageTypePanel));
         else if (typePanel.getSelection().equals("Input"))
         {
            if (inputPanel.getSelection().equals("Text field")) JOptionPane.showInputDialog(
                  OptionDialogFrame.this, getMessage(), "Title", getType(messageTypePanel));
            else JOptionPane.showInputDialog(OptionDialogFrame.this, getMessage(), "Title",
                  getType(messageTypePanel), null, new String[] { "Yellow", "Blue", "Red" },
                  "Blue");
         }
         else if (typePanel.getSelection().equals("Message")) JOptionPane.showMessageDialog(
               OptionDialogFrame.this, getMessage(), "Title", getType(messageTypePanel));
         else if (typePanel.getSelection().equals("Option")) JOptionPane.showOptionDialog(
               OptionDialogFrame.this, getMessage(), "Title", getType(optionTypePanel),
               getType(messageTypePanel), null, getOptions(), getOptions()[0]);
      }
   }

   public static final int DEFAULT_WIDTH = 600;
   public static final int DEFAULT_HEIGHT = 400;
```

```
   private ButtonPanel typePanel;
   private ButtonPanel messagePanel;
   private ButtonPanel messageTypePanel;
   private ButtonPanel optionTypePanel;
   private ButtonPanel optionsPanel;
   private ButtonPanel inputPanel;

   private String messageString = "Message";
   private Icon messageIcon = new ImageIcon("blue-ball.gif");
   private Object messageObject = new Date();
   private Component messageComponent = new SampleComponent();
}

/**
 * A component with a painted surface
 */

class SampleComponent extends JComponent
{
   public void paintComponent(Graphics g)
   {
      Graphics2D g2 = (Graphics2D) g;
      Rectangle2D rect = new Rectangle2D.Double(0, 0, getWidth() - 1, getHeight() - 1);
      g2.setPaint(Color.YELLOW);
      g2.fill(rect);
      g2.setPaint(Color.BLUE);
      g2.draw(rect);
   }

   public Dimension getPreferredSize()
   {
      return new Dimension(10, 10);
   }
}
```

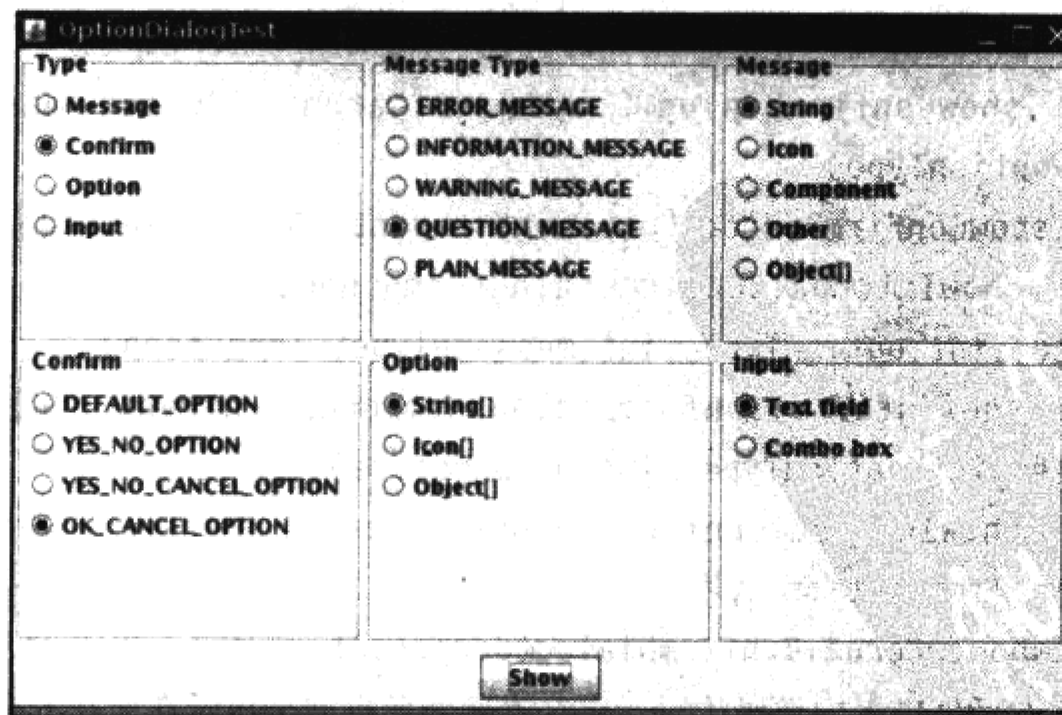


图9-37 OptionDialogTest程序

API javax.swing.JOptionPane 1.2

- `static void showMessageDialog(Component parent, Object message, String title, int messageType, Icon icon)`
- `static void showMessageDialog(Component parent, Object message, String title, int messageType)`
- `static void showMessageDialog(Component parent, Object message)`
- `static void showInternalMessageDialog(Component parent, Object message, String title, int messageType, Icon icon)`
- `static void showInternalMessageDialog(Component parent, Object message, String title, int messageType)`
- `static void showInternalMessageDialog(Component parent, Object message)`

  显示一个消息对话框或者一个内部消息对话框（内部对话框完全显示在所在的框架内）。

  参数：

parent	父组件（可以为null）。
message	显示在对话框中的消息（可以是字符串、图标、组件或者一个这些类型的数组）。
title	对话框标题栏中的字符串。
messageType	取值为 ERROR_MESSAGE、INFORMATION_MESSAGE、WARNING_MESSAGE、QUESTION_MESSAGE、PLAIN_MESSAGE之一。
icon	用于替代标准图标的图标。

- `static int showConfirmDialog(Component parent, Object message, String title, int optionType, int messageType, Icon icon)`
- `static int showConfirmDialog(Component parent, Object message, String title, int optionType, int messageType)`
- `static int showConfirmDialog(Component parent, Object message, String title, int optionType)`
- `static int showConfirmDialog(Component parent, Object message)`
- `static int showInternalConfirmDialog(Component parent, Object message, String title, int optionType, int messageType, Icon icon)`
- `static int showInternalConfirmDialog(Component parent, Object message, String title, int optionType, int messageType)`
- `static int showInternalConfirmDialog(Component parent, Object message, String title, int optionType)`
- `static int showInternalConfirmDialog(Component parent, Object message)`

  显示一个确认对话框或者内部确认对话框（内部对话框完全显示在所在的框架内）。返回用户选择的选项（取值为OK_OPTION, CANCEL_OPTION, YES_OPTION, NO_OPTION）；如

果用户关闭对话框将返回CLOSED_OPTION。

参数：parent 父组件（可以为null）。

message 显示在对话框中的消息（可以是字符串、图标、组件或者一个这些类型的数组）。

title 对话框标题栏中的字符串。

messageType 取值为ERROR_MESSAGE、INFORMATION_MESSAGE、WARNING_MESSAGE、QUESTION_MESSAGE、PLAIN_MESSAGE之一。

optionType 取值为DEFAULT_OPTION、YES_NO_OPTION、YES_NO_CANCEL_OPTION、OK_CANCEL_OPTION之一。

icon 用于替代标准图标的图标。

- `static int showOptionDialog(Component parent, Object message, String title, int optionType, int messageType, Icon icon, Object[] options, Object default)`
- `static int showInternalOptionDialog(Component parent, Object message, String title, int optionType, int messageType, Icon icon, Object[] options, Object default)`

显示一个选项对话框或者内部选项对话框（内部对话框完全显示在所在的框架内）。返回用户选择的选项索引；如果用户取消对话框返回CLOSED_OPTION。

参数：parent 父组件（可以为null）。

message 显示在对话框中的消息（可以是字符串，图标，组件或者一个这些类型的数组）。

title 对话框标题栏中的字符串。

messageType 取值为ERROR_MESSAGE、INFORMATION_MESSAGE、WARNING_MESSAGE、QUESTION_MESSAGE、PLAIN_MESSAGE之一。

optionType 取值为DEFAULT_OPTION、YES_NO_OPTION、YES_NO_CANCEL_OPTION、OK_CANCEL_OPTION之一。

icon 用于替代标准图标的图标。

options 一组选项（可以是字符串、图标或者组件）。

default 呈现给用户的默认值。

- `static Object showInputDialog(Component parent, Object message, String title, int messageType, Icon icon, Object[] values, Object default)`
- `static String showInputDialog(Component parent, Object message, String title, int messageType)`
- `static String showInputDialog(Component parent, Object message)`
- `static String showInputDialog(Object message)`
- `static String showInputDialog(Component parent, Object message, Object`

default) 1.4
- static String showInputDialog(Object message, Object default) 1.4
- static Object showInternalInputDialog(Component parent, Object message, String title, int messageType, Icon icon, Object[] values, Object default)
- static String showInternalInputDialog(Component parent, Object message, String title, int messageType)
- static String showInternalInputDialog(Component parent, Object message)

显示一个输入对话框或者内部输入对话框（内部对话框完全显示在所在的框架内）。返回用户输入的字符串；如果用户取消对话框返回null。

参数：

parent	父组件（可以为null）。
message	显示在对话框中的消息（可以是字符串、图标、组件或者一个这些类型的数组）。
title	对话框标题栏中的字符串。
messageType	取值为ERROR_MESSAGE、INFORMATION_MESSAGE、WARNING_MESSAGE、QUESTION_MESSAGE、PLAIN_MESSAGE之一。
icon	用于替代标准图标的图标。
values	在组合框中显示的一组值。
default	呈现给用户的默认值。

9.7.2 创建对话框

在上一节中，介绍了如何使用JOptionPane来显示一个简单的对话框。本节将讲述如何手工地创建这样一个对话框。

图9-38显示了一个典型的模式对话框。当用户点击About按钮时就会显示这样一个程序信息对话框。

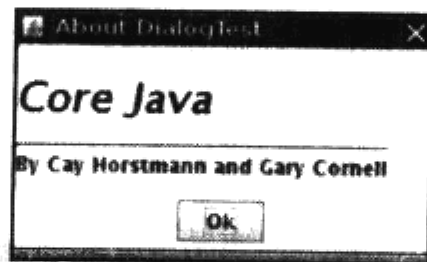


图9-38 About对话框

要想创建一个对话框，需要从JDialog派生一个类。这与应用程序窗口派生于JFrame的过程完全一样。具体过程如下：

1）在对话框构造器中，调用超类JDialog的构造器。

2）添加对话框的用户界面组件。

3）添加事件处理器。

4）设置对话框的大小。

在调用超类构造器时，需要提供*拥有者框架*（owner frame）、对话框标题及特征。

拥有者框架控制对话框的显示位置，如果将拥有者标识为null，那么对话框将由一个隐藏框架所拥有。

特征将指定对话框处于显示状态时，应用程序中其他窗口是否被锁住。无模式对话框不会锁住其他窗口，而有模式对话框将锁住应用程序中的所有其他窗口（除对话框的子窗口外）。

用户经常使用的工具栏就是无模式对话框，换句话说，如果想强迫用户在继续操作之前提供一些必要的信息就应该使用模式对话框。

注释：在Java SE 6.中，有两个额外的特征类型。文档－模式对话框将阻塞所有属于相同"文档"的窗口。更准确地说，是所有作为对话框的具有相同无父根窗口的窗口。这样解决了帮助系统的问题。在早期的版本中，当弹出一个模式对话框时，用户不可能与帮助窗口结合。工具箱对话框阻塞了所有来自于相同"工具箱"的窗口。工具箱是一个运行于多个应用的Java程序，例如，浏览器中的小应用程序引擎。有关更加详细的内容请参看网站：http://java.sun.com/developer/technicalArticles/J2SE/Desktop/javase6/modality。

下面是一个对话框的例子：

```
public AboutDialog extends JDialog
{
   public AboutDialog(JFrame owner)
   {
      super(owner, "About DialogTest", true);
      add(new JLabel(
         "<html><h1><i>Core Java</i></h1><hr>By Cay Horstmann and Gary Cornell</html>"),
         BorderLayout.CENTER);

      JPanel panel = new JPanel();
      JButton ok = new JButton("Ok");

      ok.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               setVisible(false);
            }
         });

      panel.add(ok);
      add(panel, BorderLayout.SOUTH);

      setSize(250, 150);
   }
}
```

正如看到的，构造器添加了用户界面组件，在本例中添加是标签和按钮，并且为按钮设置了处理器，然后还设置了对话框的大小。

要想显示对话框，需要建立一个新的对话框对象，并让它可见：

```
JDialog dialog = new AboutDialog(this);
dialog.setVisible(true);
```

实际上，在下面的示例代码中，只建立了一次对话框，无论何时用户点击About按钮，都可以重复使用它。

```
if (dialog == null) // first time
   dialog = new AboutDialog(this);
dialog.setVisible(true);
```

当用户点击OK按钮时，该对话框将被关闭。下面是在OK按钮的事件处理器中的处理代码：

```
ok.addActionListener(new
   ActionListener()
   {
      public void actionPerformed(ActionEvent event)
      {
         setVisible(false);
      }
   });
```

当用户点击Close按钮关闭对话框时，对话框就被隐藏起来。与JFrame一样，可以覆盖setDefaultCloseOperation方法来改变这个行为。

例9-15 是About对话框的测试程序代码。

例9-15 DialogTest.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * @version 1.33 2007-06-12
 * @author Cay Horstmann
 */
public class DialogTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               DialogFrame frame = new DialogFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * A frame with a menu whose File->About action shows a dialog.
 */
class DialogFrame extends JFrame
{
   public DialogFrame()
   {
      setTitle("DialogTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // construct a File menu

      JMenuBar menuBar = new JMenuBar();
      setJMenuBar(menuBar);
      JMenu fileMenu = new JMenu("File");
      menuBar.add(fileMenu);

```

```
        // add About and Exit menu items

        // The About item shows the About dialog

        JMenuItem aboutItem = new JMenuItem("About");
        aboutItem.addActionListener(new ActionListener()
           {
              public void actionPerformed(ActionEvent event)
              {
                 if (dialog == null) // first time
                 dialog = new AboutDialog(DialogFrame.this);
                 dialog.setVisible(true); // pop up dialog
              }
           });
        fileMenu.add(aboutItem);

        // The Exit item exits the program

        JMenuItem exitItem = new JMenuItem("Exit");
        exitItem.addActionListener(new ActionListener()
           {
              public void actionPerformed(ActionEvent event)
              {
                 System.exit(0);
              }
           });
        fileMenu.add(exitItem);
     }

     public static final int DEFAULT_WIDTH = 300;
     public static final int DEFAULT_HEIGHT = 200;

     private AboutDialog dialog;
  }

  /**
   * A sample modal dialog that displays a message and waits for the user to click the Ok button.
   */
  class AboutDialog extends JDialog
  {
     public AboutDialog(JFrame owner)
     {
        super(owner, "About DialogTest", true);

        // add HTML label to center

        add(
              new JLabel(
                 "<html><h1><i>Core Java</i></h1><hr>By Cay Horstmann and Gary Cornell</html>"),
              BorderLayout.CENTER);

        // Ok button closes the dialog

        JButton ok = new JButton("Ok");
        ok.addActionListener(new ActionListener()
           {
              public void actionPerformed(ActionEvent event)
```

```
            {
               setVisible(false);
            }
         });

      // add Ok button to southern border

      JPanel panel = new JPanel();
      panel.add(ok);
      add(panel, BorderLayout.SOUTH);

      setSize(250, 150);
   }
}
```

API javax.swing.JDialog 1.2

- `public JDialog(Frame parent, String title, boolean modal)`

  构造一个对话框。在没有明确地让对话框显示之前，它是不可见的。

  参数：parent　对话框拥有者的框架

  title　对话框的标题

  modal　True代表模式对话框（模式对话框阻隔其他窗口的输入）

9.7.3 数据交换

使用对话框最通常的目的是获取用户的输入信息。在前面已经看到，构造对话框对象非常简单：首先初始化数据，然后调用setVisible(true)就会在屏幕上显示对话框。现在，看看如何将数据传入传出对话框。

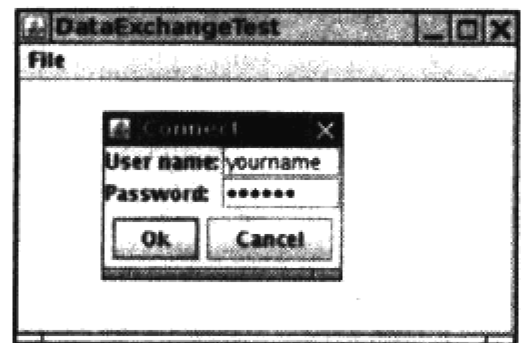


图9-39　密码对话框

看一下如图9-39所示的对话框，可以用来获得用户名和用户密码以便连接某些在线服务。

对话框应该提供设置默认数据的方法。例如，示例程序中的PasswordChooser类提供了一个setUser方法，用来将默认值放到下面的字段中：

```
public void setUser(User u)
{
   username.setText(u.getName());
}
```

一旦设置了默认值（如果需要），就可以调用setVisible(true)让对话框显示在屏幕上。

然后用户输入信息，点击OK或者Cancel按钮。这两个按钮的事件处理器都会调用setVisible(false)终止对setVisible(true)的调用。另外，用户也可以选择关闭对话框。如果没有为对话框安装窗口监听器，就会执行默认的窗口结束操作，即对话框变为不可见，这也中止了对setVisible(true)的调用。

重要的问题是在用户解除这个对话框之前，一直调用setVisible(true)阻塞。这样易于实现

模式对话框。

希望知道用户是接收对话框，还是取消对话框。在示例代码中设置了OK标志，在对话框显示之前是false。只有OK按钮的事件处理器可以将它设置为true。这样，就可以获得对话框中的用户输入。

注释：无模式对话框数据传输就没有那么简单了。当无模式对话框显示时，调用setVisible(true)并不阻塞，在对话框显示时，其他程序仍继续运行。如果用户选择了无模式对话框中的一项，并点击OK，对话框就会将一个事件发送给程序中的某个监听器。

示例程序中还包含另外一个很有用的改进。在构造一个JDialog对象时，需要指定拥有者框架。但是，在很多情况下，一个对话框可能会有多个拥有者框架，所以最好在准备显示对话框时再确定拥有者框架，而不是在构造PasswordChooser对象时。

有一个技巧是让PasswordChooser扩展于JPanel，而不是扩展于JDialog，在showDialog方法中立即建立JDialog对象：

```
public boolean showDialog(Frame owner, String title)
{
   ok = false;

   if (dialog == null || dialog.getOwner() != owner)
   {
      dialog = new JDialog(owner, true);
      dialog.add(this);
      dialog.pack();
   }

   dialog.setTitle(title);
   dialog.setVisible(true);
   return ok;
}
```

注意，让owner等于null比较安全。

可以再做进一步的改进。有时，拥有者框架并不总是可用的。利用任意的parent组件可以很容易地得到它。如下所示：

```
Frame owner;
if (parent instanceof Frame)
   owner = (Frame) parent;
else
   owner = (Frame) SwingUtilities.getAncestorOfClass(Frame.class, parent);
```

在示例程序中使用了改进的代码。JOptionPane类也使用了上面的机制。

很多对话框都有*默认按钮*。如果用户按下一个触发器键（在大多数“观感”实现中是ENTER）就自动地选择了它。默认按钮通常用加粗的轮廓给予特别的标识。

可以在对话框的*根窗格*（root pane）中设置默认按钮：

```
dialog.getRootPane().setDefaultButton(okButton);
```

如果按照前面的建议，在一个面板中布局对话框就必须特别小心。在包装面板进入对话框后再设置默认按钮。面板本身没有根窗格。

例9-16是一个描述对话框输入和输出数据流的完整程序。

例9-16 **DataExchangeTest.java**

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * @version 1.33 2007-06-12
 * @author Cay Horstmann
 */
public class DataExchangeTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               DataExchangeFrame frame = new DataExchangeFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * A frame with a menu whose File->Connect action shows a password dialog.
 */
class DataExchangeFrame extends JFrame
{
   public DataExchangeFrame()
   {
      setTitle("DataExchangeTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // construct a File menu

      JMenuBar mbar = new JMenuBar();
      setJMenuBar(mbar);
      JMenu fileMenu = new JMenu("File");
      mbar.add(fileMenu);

      // add Connect and Exit menu items

      JMenuItem connectItem = new JMenuItem("Connect");
      connectItem.addActionListener(new ConnectAction());
      fileMenu.add(connectItem);

      // The Exit item exits the program

      JMenuItem exitItem = new JMenuItem("Exit");
      exitItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
```

```
            {
               System.exit(0);
            }
         });
      fileMenu.add(exitItem);

      textArea = new JTextArea();
      add(new JScrollPane(textArea), BorderLayout.CENTER);
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;

   private PasswordChooser dialog = null;
   private JTextArea textArea;

   /**
    * The Connect action pops up the password dialog.
    */

   private class ConnectAction implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         // if first time, construct dialog

         if (dialog == null) dialog = new PasswordChooser();

         // set default values
         dialog.setUser(new User("yourname", null));

         // pop up dialog
         if (dialog.showDialog(DataExchangeFrame.this, "Connect"))
         {
            // if accepted, retrieve user input
            User u = dialog.getUser();
            textArea.append("user name = " + u.getName() + ", password = "
                  + (new String(u.getPassword())) + "\n");
         }
      }
   }
}

/**
 * A password chooser that is shown inside a dialog
 */
class PasswordChooser extends JPanel
{
   public PasswordChooser()
   {
      setLayout(new BorderLayout());

      // construct a panel with user name and password fields

      JPanel panel = new JPanel();
      panel.setLayout(new GridLayout(2, 2));
      panel.add(new JLabel("User name:"));
```

```
      panel.add(username = new JTextField(""));
      panel.add(new JLabel("Password:"));
      panel.add(password = new JPasswordField(""));
      add(panel, BorderLayout.CENTER);

      // create Ok and Cancel buttons that terminate the dialog

      okButton = new JButton("Ok");
      okButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               ok = true;
               dialog.setVisible(false);
            }
         });

      JButton cancelButton = new JButton("Cancel");
      cancelButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               dialog.setVisible(false);
            }
         });

      // add buttons to southern border

      JPanel buttonPanel = new JPanel();
      buttonPanel.add(okButton);
      buttonPanel.add(cancelButton);
      add(buttonPanel, BorderLayout.SOUTH);
   }

   /**
    * Sets the dialog defaults.
    * @param u the default user information
    */
   public void setUser(User u)
   {
      username.setText(u.getName());
   }

   /**
    * Gets the dialog entries.
    * @return a User object whose state represents the dialog entries
    */
   public User getUser()
   {
      return new User(username.getText(), password.getPassword());
   }

   /**
    * Show the chooser panel in a dialog
    * @param parent a component in the owner frame or null
    * @param title the dialog window title
    */
```

```
   public boolean showDialog(Component parent, String title)
   {
      ok = false;

      // locate the owner frame

      Frame owner = null;
      if (parent instanceof Frame) owner = (Frame) parent;
      else owner = (Frame) SwingUtilities.getAncestorOfClass(Frame.class, parent);

      // if first time, or if owner has changed, make new dialog

      if (dialog == null || dialog.getOwner() != owner)
      {
         dialog = new JDialog(owner, true);
         dialog.add(this);
         dialog.getRootPane().setDefaultButton(okButton);
         dialog.pack();
      }

      // set title and show dialog

      dialog.setTitle(title);
      dialog.setVisible(true);
      return ok;
   }

   private JTextField username;
   private JPasswordField password;
   private JButton okButton;
   private boolean ok;
   private JDialog dialog;
}

/**
 * A user has a name and password. For security reasons, the password is stored as a char[],
 * not a String.
 */
class User
{
   public User(String aName, char[] aPassword)
   {
      name = aName;
      password = aPassword;
   }

   public String getName()
   {
      return name;
   }

   public char[] getPassword()
   {
      return password;
   }

   public void setName(String aName)
```

```
    {
        name = aName;
    }

    public void setPassword(char[] aPassword)
    {
        password = aPassword;
    }

    private String name;
    private char[] password;
}
```

API javax.swing.SwingUtilities 1.2

- `Container getAncestorOfClass(Class c, Component comp)`

  返回给定组件的最先的父容器。这个组件属于给定的类或者其子类之一。

API javax.swing.JComponent 1.2

- `JRootPane getRootPane()`

  获得最靠近这个组件的根窗格，如果这个组件的祖先没有根窗格返回null。

API javax.swing.JRootPane 1.2

- `void setDefaultButton(JButton button)`

  设置根窗格的默认按钮。要想禁用默认按钮，需要将参数设置为null。

API javax.swing.JButton 1.2

- `boolean isDefaultButton()`

  如果这个按钮是它的根窗格的默认按钮，返回true。

9.7.4 文件对话框

当编写应用程序时，通常希望可以打开和保存文件。一个好的文件对话框应该可以显示文件和目录，可以让用户浏览文件系统，这是很难编写的。人们肯定不愿意从头做起。很幸运，Swing中提供了JFileChooser类，它可以显示一个文件对话框，其外观与本地应用程序中使用的文件对话框基本一样。JFileChooser是一个模式对话框。注意，JFileChooser类并不是JDialog类的子类。需要调用showOpenDialog，而不是调用setVisible(true)显示打开文件的对话框，或者调用showSaveDialog显示保存文件的对话框。接收文件的按钮被自动地标签为Open或者Save。也可以调用showDialog方法为按钮设定标签。图9-40是文件选择对话框的样例。

下面是建立文件对话框并且获取用户选择信息的步骤：

1）建立一个JFileChooser对象。与JDialog类的构造器不同，它不需要指定父组件。允许在多个框架中重用一个文件选择器。例如：

```
JFileChooser chooser = new JFileChooser();
```

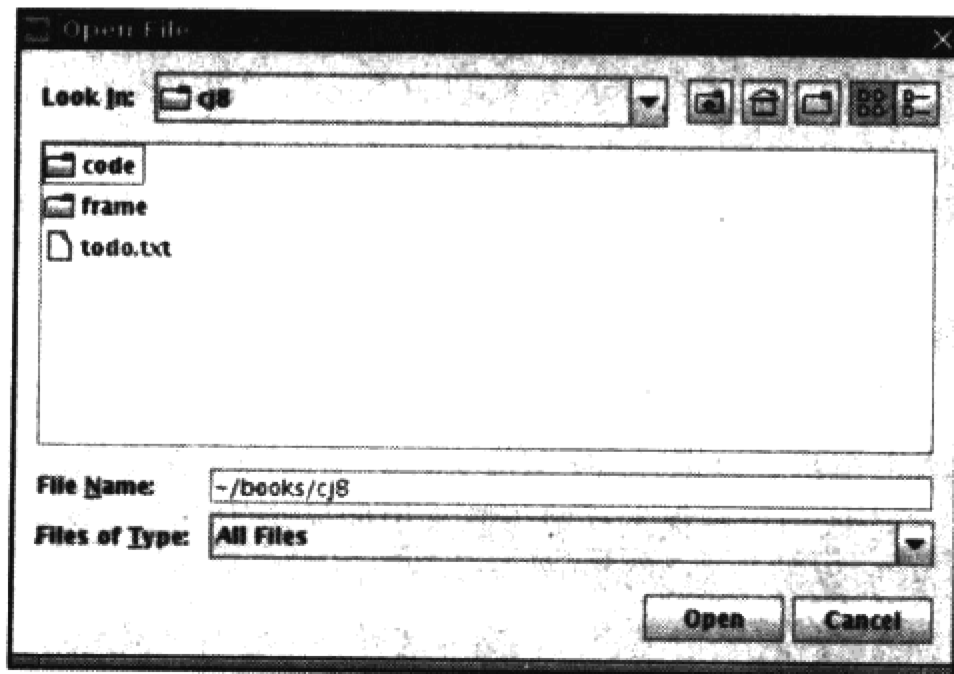


图9-40 文件选择对话框

提示： 重用一个文件选择器对象是一个很好的想法，其原因是JFileChooser的构造器相当耗费时间。特别是在Windows上，用户映射了很多网络驱动器的情况下。

2）调用setCurrentDirectory方法设置当前目录。例如，使用当前的工作目录：

```
chooser.setCurrentDirectory(new File("."));
```

需要提供一个File对象。File对象将在第12章中详细地介绍。这里只需要知道构造器File（String fileName）能够将一个文件或目录名转化为一个File对象即可。

3）如果有一个想要作为用户选择的默认文件名，可以使用setSelectedFile方法进行指定：

```
chooser.setSelectedFile(new File(filename));
```

4）如果允许用户在对话框中选择多个文件，需要调用setMultiSelectionEnabled方法。当然，这是可选的。

```
chooser.setMultiSelectionEnabled(true);
```

5）如果想让对话框仅显示某一种类型的文件（如，所有扩展名为.gif的文件），需要设置文件过滤器，稍后将会进行讨论。

6）在默认情况下，用户在文件选择器中只能选择文件。如果希望选择目录，需要调用setFileSelectionMode方法。参数值为：JFileChooser.FILES_ONLY（默认值），JFileChooser.DIRECTORIES_ONLY或者JFileChooser.FILES_AND_DIRECTORIES。

7）调用showOpenDialog 或者showSaveDialog方法显示对话框。必须为这些调用提供父组件：

```
int result = chooser.showOpenDialog(parent);
```

或者

```
int result = chooser.showSaveDialog(parent);
```

这些调用的区别是“确认按钮”的标签不同。点击“确认按钮”将完成文件选择。也可以调用showDialog方法，并将一个显式的文本传递给确认按钮：

```
int result = chooser.showDialog(parent, "Select");
```

仅当用户确认、取消或者离开对话框时才返回调用。返回值可以是JFileChooser. APPROVE_OPTION、JFileChooser.CANCEL_OPTION或者JFileChooser.ERROR_OPTION。

8）调用getSelectedFile() 或者 getSelectedFiles() 方法获取用户选择的一个或多个文件。这些方法将返回一个文件对象或者一组文件对象。如果需要知道文件对象名时，可以调用getPath方法。例如：

```
String filename = chooser.getSelectedFile().getPath();
```

在大多数情况下，这些过程比较简单。使用文件对话框的主要困难在于指定用户需要选择的文件子集。例如，假定用户应该选择GIF图像文件。后面的文件选择器就应该只显示扩展名为.gif的文件，并且，还应该为用户提供反馈信息来说明显示的特定文件类别，如"GIF图像"。然而，情况有可能会更加复杂。如果用户应该选择JPFG图像文件，扩展名就可以是.jpg或者.jpeg。与重新编码实现这种复杂情况相比，文件选择器的设计者提供了一种更好的机制：若想限制显示的文件，需要创建一个实现了抽象类javax.swing.filechooser.FileFilter的对象。文件选择器将每个文件传递给文件过滤器，只有文件过滤器接受的文件才被最终显示出来。

在编写本书的时候，有两个子类可用：可以接受所有文件的默认过滤器和可以接受给定扩展名的所有文件的过滤器。其实，设计专用文件过滤器非常简单，只要实现FileFilter超类中的两个方法即可：

```
public boolean accept(File f);
public String getDescription();
```

第一个方法检测是否应该接受一个文件，第二个方法返回显示在文件选择器对话框种显示的文件类型的描述信息。

注释：*在java.io包中有一个无关的FileFilter接口，其中只包含一个方法：boolean accept(File f)。File类中的listFiles方法利用它显示目录中的文件。我们不知道Swing的设计者为什么不扩展这个接口，可能是因为Java类库过于复杂，致使Sun程序员也不太熟悉所有的标准类和接口了。*

*需要解决同时导入javax.io包和javax.swing.filechooser包带来的名称冲突问题。最简单的方法是导入javax.swing.filechooser.Filefilter，而不要导入javax.swing.filechooser.**

一旦有了文件过滤器对象，就可以调用JFileChooser类中的setFileFilter方法，将这个对象安装到文件选择器对象中：

```
chooser.setFileFilter(new FileNameExtensionFilter("Image files", "gif", "jpg");
```

可以为一个文件选择器安装多个过滤器：

```
chooser.addChoosableFileFilter(filter1);
chooser.addChoosableFileFilter(filter2);
. . .
```

用户可以从文件对话框底部的组合框中选择过滤器。在默认情况下，All files过滤器总是显示在组合框中。这是一个很好的主意，特别是在使用这个程序的用户需要选择一个具有非标准扩展名的文件时。然而，如果你想放弃All files过滤器，需要调用：

```
chooser.setAcceptAllFileFilterUsed(false)
```

警告：如果为加载和保存不同类型的文件重用一个文件选择器，就需要调用：

```
chooser.resetChoosableFilters()
```

这样可以在添加新文件过滤器之前清除旧文件过滤器。

最后，可以通过为文件选择器显示的每个文件提供特定的图标和文件描述来定制文件选择器。这需要应用一个扩展于javax.swing.filechooser包中的FileView类的对象。这是一个高级技巧。在通常情况下，不需要提供文件视图——可插观感会提供。然而，如果想让某种特定的文件类型显示不同的图标，就需要安装自己的文件视图。这要扩展FileView并实现下面5五个方法：

```
Icon getIcon(File f);
String getName(File f);
String getDescription(File f);
String getTypeDescription(File f);
Boolean isTraversable(File f);
```

然后，调用setFileView方法将文件视图安装到文件选择器中。

文件选择器为每个希望显示的文件或目录调用这些方法。如果方法返回的图标、名字或描述信息为null，那么文件选择器将会构造当前观感的默认文件视图。这样处理很好，其原因是这样只需处理具有不同显示的文件类型。

文件选择器调用isTraversable方法来决定是否在用户点击一个目录的时候打开这个目录。请注意，这个方法返回一个Boolean对象，而不是boolean值。看起来似乎有点怪，但实际上很方便——如果需要使用默认的视图，则返回null。文件选择器将会使用默认的文件视图。换句话说，这个方法返回的Boolean对象能给出下面三种选择：真(Boolean.TRUE)，假(Boolean.FALSE)和不关心 (null)。

在示例中包含了一个简单的文件视图类。当文件匹配文件过滤器时，这个类将会显示一个特定的图标。可以利用这个类为所有的图像文件显示一个调色图标。

```
class FileIconView extends FileView
{
   public FileIconView(FileFilter aFilter, Icon anIcon)
   {
      filter = aFilter;
      icon = anIcon;
   }

   public Icon getIcon(File f)
   {
      if (!f.isDirectory() && filter.accept(f))
         return icon;
      else return null;
   }
   private FileFilter filter;
   private Icon icon;
}
```

可以调用setFileView方法将这个文件视图安装到文件选择器中：

```
chooser.setFileView(new FileIconView(filter,
```

```
new ImageIcon("palette.gif")));
```

文件选择器会在通过filter的所有文件旁边显示调色板图标，并且使用默认的文件视图来显示所有其他的文件。很自然，我们可以使用与文件选择器设定的一样的过滤器。

提示：可以在JDK的demo/jfc/FileChooserDemo目录下找到更实用的ExampleFileView类。它可以将图标和描述信息与所有扩展名关联起来。

最后，可以通过添加一个附件组件来定制文件对话框。例如，图9-41在文件列表旁边显示了一个预览附件。这个附件显示了当前选择文件的预览信息。

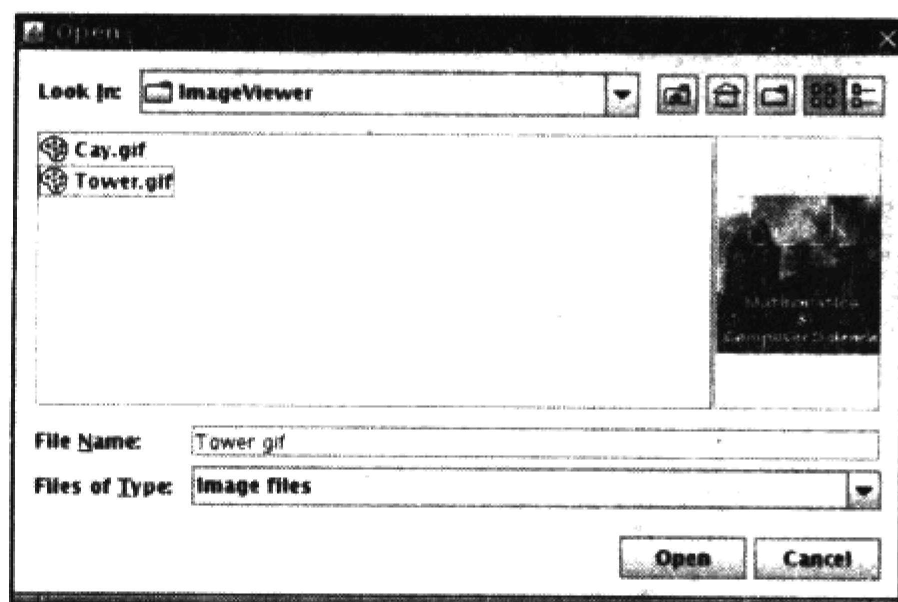


图9-41 带预览附件的文件对话框

附件可以是任何Swing组件。在这个示例中，扩展JLabel类，并将图标设置为所选的图像文件的压缩拷贝。

```
class ImagePreviewer extends JLabel
{
   public ImagePreviewer(JFileChooser chooser)
   {
      setPreferredSize(new Dimension(100, 100));
      setBorder(BorderFactory.createEtchedBorder());
   }

   public void loadImage(File f)
   {
      ImageIcon icon = new ImageIcon(f.getPath());
      if(icon.getIconWidth() > getWidth())
         icon = new ImageIcon(icon.getImage().getScaledInstance(
            getWidth(), -1, Image.SCALE_DEFAULT));
      setIcon(icon);
      repaint();
   }
}
```

这里还有一个挑战，即需要在用户选择不同的文件时更新预览图像。文件选择器使用了Java Beans机制。当它的属性发生变化时，文件选择器就会通知相关的监听器。被选择文件是

一个属性，可以通过安装PropertyChangeListener监听它。本书将在卷II中讨论这个机制。下面这段代码可以用来捕捉通知：

```
chooser.addPropertyChangeListener(new
   PropertyChangeListener()
   {
      public void propertyChange(PropertyChangeEvent event)
      {
         if (event.getPropertyName() == JFileChooser.SELECTED_FILE_CHANGED_PROPERTY)
         {
            File newFile = (File) event.getNewValue()
             // update the accessory
             . . .
         }
      }
});
```

在这个示例中，将这段代码添加到ImagePreviewer构造器中。

例9-17对第2章中的ImageViewer程序做了一定的修改。通过自定义的文件视图和预览附件文件增强了文件选择器的功能。

例9-17 FileChooserTest.java

```
import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.util.*;
import java.io.*;
import javax.swing.*;
import javax.swing.filechooser.*;
import javax.swing.filechooser.FileFilter;

/**
 * @version 1.23 2007-06-12
 * @author Cay Horstmann
 */
public class FileChooserTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               ImageViewerFrame frame = new ImageViewerFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * A frame that has a menu for loading an image and a display area for the loaded image.
 */
```

```
class ImageViewerFrame extends JFrame
{
   public ImageViewerFrame()
   {
      setTitle("FileChooserTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // set up menu bar
      JMenuBar menuBar = new JMenuBar();
      setJMenuBar(menuBar);

      JMenu menu = new JMenu("File");
      menuBar.add(menu);

      JMenuItem openItem = new JMenuItem("Open");
      menu.add(openItem);
      openItem.addActionListener(new FileOpenListener());

      JMenuItem exitItem = new JMenuItem("Exit");
      menu.add(exitItem);
      exitItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               System.exit(0);
            }
         });

      // use a label to display the images
      label = new JLabel();
      add(label);

      // set up file chooser
      chooser = new JFileChooser();

      // accept all image files ending with .jpg, .jpeg, .gif
      /*
      final ExtensionFileFilter filter = new ExtensionFileFilter();
      filter.addExtension("jpg");
      filter.addExtension("jpeg");
      filter.addExtension("gif");
      filter.setDescription("Image files");
      */
      FileNameExtensionFilter filter = new FileNameExtensionFilter(
            "Image files", "jpg", "jpeg", "gif");
      chooser.setFileFilter(filter);

      chooser.setAccessory(new ImagePreviewer(chooser));

      chooser.setFileView(new FileIconView(filter, new ImageIcon("palette.gif")));
   }

   /**
    * This is the listener for the File->Open menu item.
    */
   private class FileOpenListener implements ActionListener
   {
```

```
      public void actionPerformed(ActionEvent event)
      {
         chooser.setCurrentDirectory(new File("."));

         // show file chooser dialog
         int result = chooser.showOpenDialog(ImageViewerFrame.this);

         // if image file accepted, set it as icon of the label
         if (result == JFileChooser.APPROVE_OPTION)
         {
            String name = chooser.getSelectedFile().getPath();
            label.setIcon(new ImageIcon(name));
         }
      }
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 400;

   private JLabel label;
   private JFileChooser chooser;
}

/**
 * A file view that displays an icon for all files that match a file filter.
 */
class FileIconView extends FileView
{
   /**
    * Constructs a FileIconView.
    * @param aFilter a file filter--all files that this filter accepts will be shown with the
    * icon. @param anIcon--the icon shown with all accepted files.
    */
   public FileIconView(FileFilter aFilter, Icon anIcon)
   {
      filter = aFilter;
      icon = anIcon;
   }

   public Icon getIcon(File f)
   {
      if (!f.isDirectory() && filter.accept(f)) return icon;
      else return null;
   }

   private FileFilter filter;
   private Icon icon;
}

/**
 * A file chooser accessory that previews images.
 */
class ImagePreviewer extends JLabel
{
   /**
    * Constructs an ImagePreviewer.
```

```
     * @param chooser the file chooser whose property changes trigger an image change in this
     * previewer
     */
   public ImagePreviewer(JFileChooser chooser)
   {
      setPreferredSize(new Dimension(100, 100));
      setBorder(BorderFactory.createEtchedBorder());

      chooser.addPropertyChangeListener(new PropertyChangeListener()
         {
            public void propertyChange(PropertyChangeEvent event)
            {
               if (event.getPropertyName() == JFileChooser.SELECTED_FILE_CHANGED_PROPERTY)
               {
                  // the user has selected a new file
                  File f = (File) event.getNewValue();
                  if (f == null)
                  {
                     setIcon(null);
                     return;
                  }

                  // read the image into an icon
                  ImageIcon icon = new ImageIcon(f.getPath());

                  // if the icon is too large to fit, scale it
                  if (icon.getIconWidth() > getWidth()) icon = new ImageIcon(icon.getImage()
                        .getScaledInstance(getWidth(), -1, Image.SCALE_DEFAULT));

                  setIcon(icon);
               }
            }
         });
   }
}
```

API javax.swing.JFileChooser 1.2

- JFileChooser()

  创建一个可用于多框架的文件选择器对话框。
- void setCurrentDirectory(File dir)

  设置文件对话框的初始目录。
- void setSelectedFile(File file)
- void setSelectedFiles(File[] file)

  设置文件对话框的默认文件选择。
- void setMultiSelectionEnabled(boolean b)

  设置或清除多选模式。
- void setFileSelectionMode(int mode)

  设置用户选择模式，只可以选择文件（默认），只可以选择目录，或者文件和目录均可

以选择。mode参数的取值可以是JFileChooser.FILES_ONLY、JFileChooser.DIRECTORIES_ONLY和JFileChooser.FILES_AND_DIRECTORIES之一。

- int showOpenDialog(Component parent)
- int showSaveDialog(Component parent)
- int showDialog(Component parent, String approveButtonText)

  显示按钮标签为Open，Save或者approveButtonText字符串的对话框，并返回APPROVE_OPTION、CANCEL_OPTION（如果用户选择取消按钮或者离开了对话框）或者ERROR_OPTION（如果发生错误）。
- File getSelectedFile()
- File[] getSelectedFiles()

  获取用户选择的一个文件或多个文件（如果用户没有选择文件，返回null）。
- void setFileFilter(FileFilter filter)

  设置文件对话框的文件过滤器。所有让filter.accept返回true 的文件都会被显示，并且将过滤器添加到可选过滤器列表中。
- void addChoosableFileFilter(FileFilter filter)

  将文件过滤器添加到可选过滤器列表中。
- void setAcceptAllFileFilterUsed(boolean b)

  在过滤器组合框中包括或者取消All files过滤器。
- void resetChoosableFileFilters()

  清除可选过滤器列表。除非All files过滤器被显式地清除，否则它仍然会存在。
- void setFileView(FileView view)

  设置一个文件视图来提供文件选择器显示信息。
- void setAccessory(JComponent component)

  设置一个附件组件。

API javax.swing.filechooser.FileFilter 1.2

- boolean accept(File f)

  如果文件选择器可以显示这个文件，返回true。
- String getDescription()

  返回这个文件过滤器的说明信息，例如，Image files (*.gif,*.jpeg)。

API javax.swing.filechooser.FileNameExtensionFiler 6

- FileNameExtensionFilter(String description, String ... extensions)

  利用给定的描述构造一个文件过滤器。这些描述限定了被接受的所有目录和文件其名称结尾的句点之后所包含的扩展字符串。

API javax.swing.filechooser.FileView 1.2

- String getName(File f)

返回文件f的文件名，或者null。通常这个方法返回f.getName()。

- `String getDescription(File f)`

  返回文件f的可读性描述，或者null。例如，如果f是HTML文档，那么这个方法有可能返回它的标题。

- `String getTypeDescription(File f)`

  返回文件f的类型的可读性描述。例如，如果f是HTML文档，那么这个方法有可能返回Hypertext document字符串。

- `Icon getIcon(File f)`

  返回文件f的图标，或者null。例如，如果f是JPEG文件，那么这个方法有可能返回简略的图标。

- `Boolean isTraversable(File f)`

  如果f是用户可以打开的目录，返回Boolean.TRUE。如果目录在概念上是复合文档，那么这个方法有可能返回false。与所有的FileView 方法一样，这个方法有可能返回null，用于表示文件选择器应该使用默认视图。

9.7.5 颜色选择器

前面曾经说过，一个高质量的文件选择器是一个很复杂的用户界面组件，人们肯定不愿意自己去编写它。许多用户界面工具包还提供了另外一些常用的对话框：选择日期/时间、货币值、字体以及颜色等。这将会带来两个方面的好处：程序员可以直接地使用这些高质量的代码而不用从头做起，并且用户通常具有使用这些选择的经验。

除了文件选择器外，Swing还提供了一种选择器——JColorChooser（如图9-42～图9-44）。可以利用这个选择器选取颜色。与JFileChooser一样，颜色选择器也是一个组件，而不是一个对话框，但是它包含了用于创建包含颜色选择器组件的对话框方法。

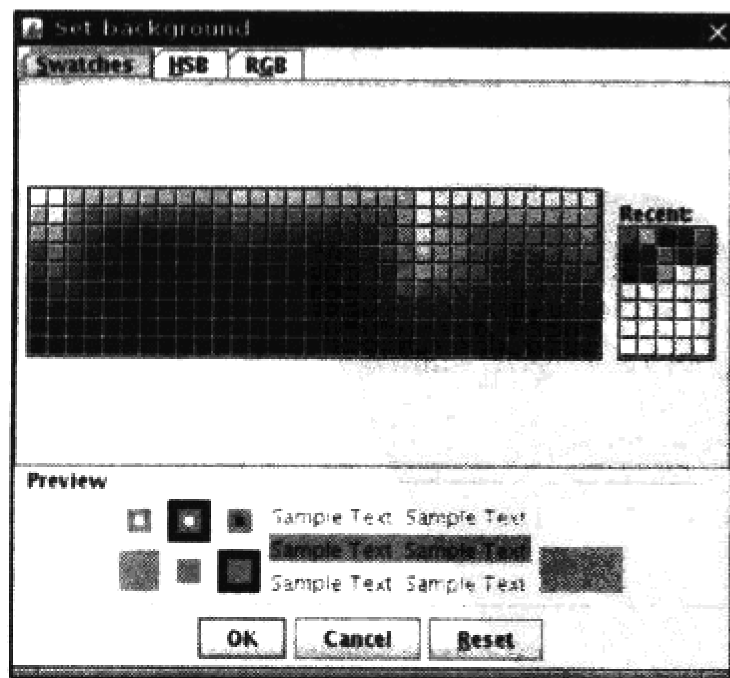


图9-42 颜色选择器的swatches窗格

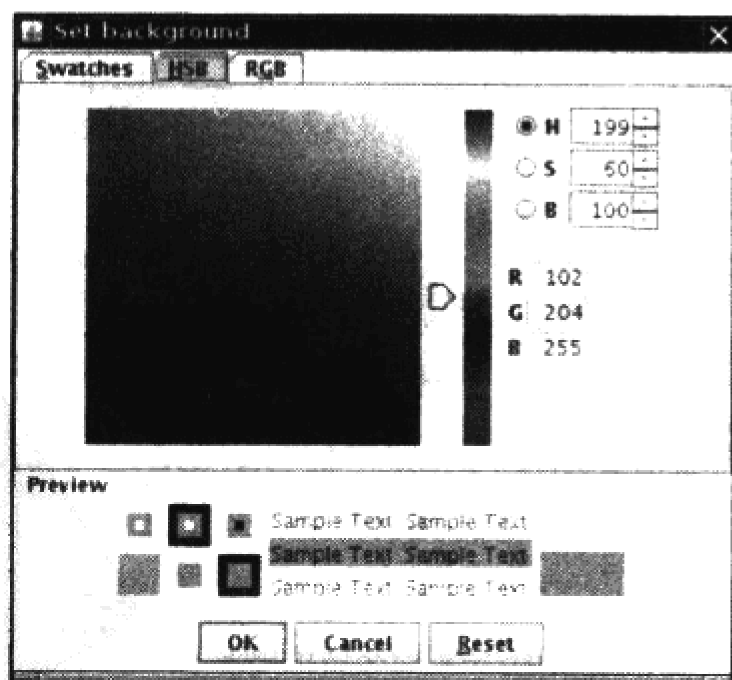


图9-43 颜色选择器的HSB窗格

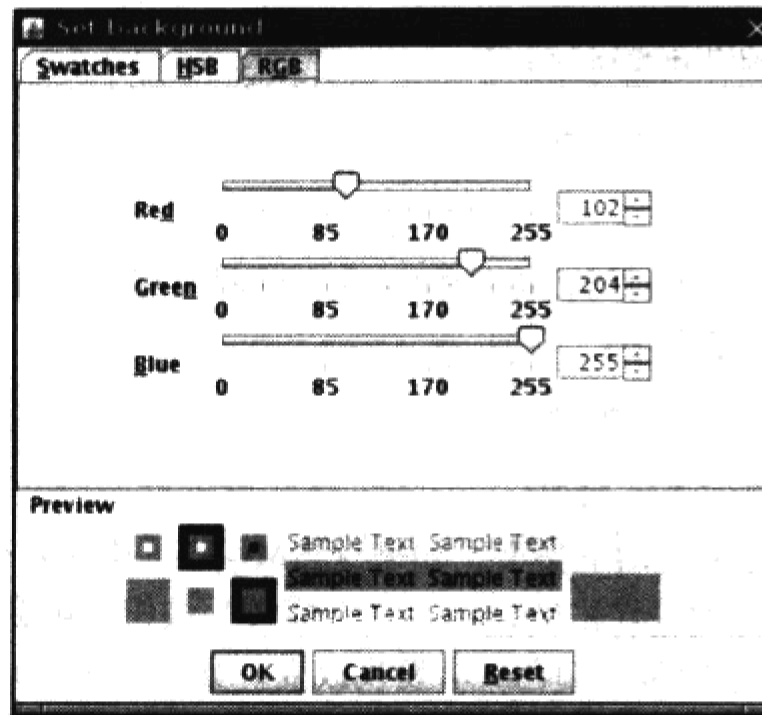


图9-44 颜色选择器的RGB窗格

下面这段代码说明了如何利用颜色选择器显示模式对话框：

```
Color selectedColor = JColorChooser.showDialog(parent,title, initialColor);
```

另外，也可以显示无模式颜色选择器对话框，需要提供：

- 一个父组件。
- 对话框的标题。
- 选择模式/无模式对话框的标志。
- 颜色选择器。
- OK和Cancel按钮的监听器（如果不需要监听器可以设置为null）。

下面这段代码将会创建一个无模式对话框。当用户按下OK键时，对话框的背景颜色就会被设成所选择的颜色。

```
chooser = new JColorChooser();
dialog = JColorChooser.createDialog(
   parent,
   "Background Color",
false /* not modal */,
chooser,
new ActionListener() // OK button listener
   {
      public void actionPerformed(ActionEvent event)
      {
         setBackground(chooser.getColor());
      }
   },
null /* no Cancel button listener */);
```

读者还可以做进一步的改进，将颜色选择立即反馈给用户。如果想要监视颜色的选择，那就需要获得选择器的选择模型并添加改变监听器：

```
chooser.getSelectionModel().addChangeListener(new
   ChangeListener()
```

```
   {
      public void stateChanged(ChangeEvent event)
      {
         do something with chooser.getColor();
      }
   });
```

在这种情况下，颜色选择器对话框提供的OK和Cancel没有什么用途。可以将颜色选择器组件直接添加到一个无模式对话框中：

```
dialog = new JDialog(parent, false /* not modal */);
dialog.add(chooser);
dialog.pack();
```

例9-18的程序展示了三种对话框类型。如果点击Model（模式）按钮，则需要选择一个颜色才能够继续后面的操作。如果点击Modaless（无模式）按钮，则会得到一个无模式对话框。这时只有点击对话框的OK按钮时，颜色才会发生改变。如果点击Immediate按钮，那将会得到一个没有按钮的无模式对话框。只要选择了对话框中的一个颜色，面板的背景就会立即更新。

例9-18 ColorChooserTest.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

/**
 * @version 1.03 2007-06-12
 * @author Cay Horstmann
 */
public class ColorChooserTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               ColorChooserFrame frame = new ColorChooserFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * A frame with a color chooser panel
 */
class ColorChooserFrame extends JFrame
{
   public ColorChooserFrame()
   {
      setTitle("ColorChooserTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
```

```

      // add color chooser panel to frame

      ColorChooserPanel panel = new ColorChooserPanel();
      add(panel);
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;
}

/**
 * A panel with buttons to pop up three types of color choosers
 */
class ColorChooserPanel extends JPanel
{
   public ColorChooserPanel()
   {
      JButton modalButton = new JButton("Modal");
      modalButton.addActionListener(new ModalListener());
      add(modalButton);

      JButton modelessButton = new JButton("Modeless");
      modelessButton.addActionListener(new ModelessListener());
      add(modelessButton);

      JButton immediateButton = new JButton("Immediate");
      immediateButton.addActionListener(new ImmediateListener());
      add(immediateButton);
   }

   /**
    * This listener pops up a modal color chooser
    */
   private class ModalListener implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         Color defaultColor = getBackground();
         Color selected = JColorChooser.showDialog(ColorChooserPanel.this, "Set background",
               defaultColor);
         if (selected != null) setBackground(selected);
      }
   }

   /**
    * This listener pops up a modeless color chooser. The panel color is changed when the user
    * clicks the Ok button.
    */
   private class ModelessListener implements ActionListener
   {
      public ModelessListener()
      {
         chooser = new JColorChooser();
         dialog = JColorChooser.createDialog(ColorChooserPanel.this, "Background Color",
               false /* not modal */, chooser, new ActionListener() // OK
               // button
```

```
                // listener
                {
                   public void actionPerformed(ActionEvent event)
                   {
                      setBackground(chooser.getColor());
                   }
                }, null /* no Cancel button listener */);
      }

      public void actionPerformed(ActionEvent event)
      {
         chooser.setColor(getBackground());
         dialog.setVisible(true);
      }

      private JDialog dialog;
      private JColorChooser chooser;
   }

   /**
    * This listener pops up a modeless color chooser. The panel color is changed immediately
    * when the user picks a new color.
    */
   private class ImmediateListener implements ActionListener
   {
      public ImmediateListener()
      {
         chooser = new JColorChooser();
         chooser.getSelectionModel().addChangeListener(new ChangeListener()
            {
               public void stateChanged(ChangeEvent event)
               {
                  setBackground(chooser.getColor());
               }
            });

         dialog = new JDialog((Frame) null, false /* not modal */);
         dialog.add(chooser);
         dialog.pack();
      }

      public void actionPerformed(ActionEvent event)
      {
         chooser.setColor(getBackground());
         dialog.setVisible(true);
      }

      private JDialog dialog;
      private JColorChooser chooser;
   }
}
```



javax.swing.JColorChooser 1.2

- JColorChooser()

构造一个初始颜色为白色的颜色选择器。

- `Color getColor()`
- `void setColor(Color c)`

获取和设置颜色选择器的当前颜色。

- `static Color showDialog(Component parent, String title, Color initialColor)`

显示包含颜色选择器的模式对话框。

参数：	
parent	对话框显示在上面的组件
title	对话框框架的标题
initialColor	颜色选择器的初始颜色

- `static JDialog createDialog(Component parent, String title, boolean modal, JColorChooser chooser, ActionListener okListener, ActionListener cancelListener)`

创建一个包含颜色选择器的对话框。

参数：	
parent	对话框显示在上面的组件
title	对话框框架的标题
modal	如果直到对话框关闭，这个调用都被阻塞，则返回true
chooser	添加到对话框中的颜色选择器
okListener, cancelListener	OK和Cancel按钮的监听器

至此将结束用户界面组件的讨论。第7章～第9章讲述了如何用Swing实现简单的图形用户界面。卷II将讨论更加高级的Swing组件和更加复杂的图形技术。

第10章　部署应用程序和applet

- ▲ JAR文件
- ▲ Java Web Start
- ▲ Applet
- ▲ 应用程序配置的存储

到目前为止，我们已经能够熟练地使用Java程序语言的大部分特性，并且对Java图形编程的基本知识也有所了解。现在准备创建提交给用户的应用程序，至此需要知道如何将这些应用程序进行打包，以便部署到用户的计算机上。传统的部署方式是使用applet，这应该归功于在Java出现的最初几年中对其给予的大肆吹捧。applet是一种特殊的Java程序，它允许通过网络下载，并可以在浏览器中运行。其目的在于让用户不再为安装软件烦恼，并且可以通过支持Java的计算机或者其他具有Internet连接的设备使用这些软件。

但是由于种种原因，applet并没有实现上述目标。因此，本章将首先介绍打包应用程序的指令，然后再论述Java Web Start机制，这是一种对基于互联网的应用程序发布的方式，最后介绍一下applet，并展示一下可能仍然会使用的环境。

本章还要讨论一下应用程序如何存储配置信息和用户参数。

10.1　JAR文件

在将应用程序进行打包时，使用者一定希望仅提供给其一个单独的文件，而不是一个含有大量类文件的目录，Java归档（JAR）文件就是为此目的而设计的。一个JAR文件既可以包含类文件，也可以包含诸如图像和声音这些其他类型的文件。此外，JAR文件是压缩的，它使用了大家熟悉的ZIP压缩格式。

> **!** **提示**：Java SE 5.0提供了一种新的压缩方案，被称为pack200，这是一种较通常的ZIP压缩算法更加有效的压缩类文件的方式。Sun声称，对类文件的压缩率接近90%。有关更加详细的信息请参看网站http://java.sun.com/javase/6/docs/technotes/guide/deployment/deployment-guide/pack200.html。

可以使用jar工具制作JAR文件（在默认的JDK安装中，位于jdk/bin目录下）。创建一个新的JAR文件应该使用的常见命令格式为：

```
jar cvf JARFileName File1 File2 . . .
```

例如：

```
jar cvf CalculatorClasses.jar *.class icon.gif
```

通常，jar命令的格式如下：

```
jar options File1 File2 . . .
```

表10-1列出了所有jar程序的可选项。它们类似于UNIX tar命令的选项。

表10-1 jar程序选项

选项	说明
c	创建一个新的或者空的存档文件并加入文件。如果指定的文件名是目录，jar程序将会对它们进行递归处理
C	暂时改变目录，例如： `jar cvf JARFileName.jar -C classes *.class` 改变classes子目录，以便增加这些类文件
e	在清单文件中创建一个条目（请参看可执行的JAR文件）
f	将JAR文件名指定为第二个命令行参数。如果没有这个参数，jar命令会将结果写到标准输出上（在创建JAR文件时）或者从标准输入中读取它（在解压或者列出JAR文件内容时）
i	建立索引文件（用于加快对大型归档的查找）
m	将一个清单文件（manifest）添加到JAR文件中。清单是对存档内容和来源的说明。每个归档有一个默认的清单文件。但是，如果想验证归档文件的内容，可以提供自己的清单文件
M	不为条目创建清单文件
t	显示内容表
u	更新一个已有的JAR文件
v	生成详细的输出结果
x	解压文件。如果提供一个或多个文件名，只解压这些文件；否则，解压所有文件
0	存储，不进行ZIP压缩

10.1.1 清单文件

除了类文件、图像和其他资源外，每个JAR文件还包含一个用于描述归档特征的清单文件(manifest)。

清单文件被命名为 MANIFEST.MF，它位于JAR文件的一个特殊META-INF 子目录中。最小的符合标准的清单文件是很简单的：

```
Manifest-Version: 1.0
```

复杂的清单文件可能包含更多条目。这些清单条目被分成多个节。第一节被称为主节(main section)。它作用于整个JAR文件。随后的条目用来指定已命名条目的属性，这些已命名的条目可以是某个文件、包或者URL。它们都必须起始于名为Name的条目。节与节之间用空行分开。例如：

```
Manifest-Version: 1.0
描述这个归档文件的行

Name: Woozle.class
描述这个文件的行

Name: com/mycompany/mypkg/
描述这个包的行
```

要想编辑清单文件，需要将希望添加到清单文件中的行放到文本文件中，然后运行：

jar cfm *JARFileName ManifestFileName* . . .

例如，要创建一个包含清单的JAR文件，应该运行：

```
jar cfm MyArchive.jar manifest.mf com/mycompany/mypkg/*.class
```

要想更新一个已有的JAR文件的清单，则需要将增加的部分放置到一个文本文件中，然后执行下列命令：

```
jar ufm MyArchive.jar manifest-additions.mf
```

注释：请参看http://java.sun.com/javase/6/docs/technotes/guides/jar获得有关JAR文件和清单文件格式的更多信息。

10.1.2 可运行JAR文件

在Java SE 6中，可以使用jar命令中的e选项指定程序的条目点，即通常需要在调用Java程序加载器时指定的类：

```
jar cvfe MyProgram.jar com.mycompany.mypkg.MainAppClass files to add
```

用户可以简单地通过下面命令来启动应用程序：

```
java -jar MyProgram.jar
```

在旧的JDK版本中，必须指定应用程序的主类，下面是这个处理命令：

```
Main-Class: com.mycompany.mypkg.MainAppClass
```

不要将扩展名.class添加到主类名中。然后运行jar命令：

```
jar cvfm MyProgram.jar mainclass.mf files to add
```

警告：清单文件的最后一行必须以换行符结束。否则，清单文件将无法被正确地读取。常见的错误是创建了一个只包含Main-Class而没有行结束符的文本文件。

根据操作系统的配置，可以通过双击JAR文件图标来启动应用程序。下面是各种操作系统的操作方式：

- 在Windows平台中，Java运行时安装器将建立一个扩展名为.jar的文件与javaw -jar命令相关联来启动文件（与java命令不同，javaw命令不打开shell窗口）。
- 在Solaris平台中，操作系统能够识别JAR文件的“魔数”格式，并用java -jar命令启动它。
- 在Mac OS X平台中，操作系统能够识别.jar 扩展名文件。当双击JAR文件时就会执行Java程序可以运行。

无论怎样，人们对JAR文件中的Java程序与本地文件有着不同的感觉。在Windows平台中，可以使用第三方的打包器工具将JAR文件转换成Windows可执行文件。打包器是一个大家熟知的扩展名为.exe的Windows程序，它可以查找和加载Java虚拟机（JVM），或者在没有找到JVM时告诉用户应该做些什么。有许多商业的和开放的原始产品，比如，JSmooth（http://jsmooth.sourceforge.net）和Launch4J（http://launch4j.sourceforge.net）。开放的原始安装生成器IzPack（http://izpack.org）还包含了一个本地加载器。有关这个问题的更加详细的信息请参看网站http://www.javalobby.org/articles/java2exe。

在Macintosh平台中，这种情形处理起来会容易一些。应用程序打包工具MRJAppBuilder可以将JAR文件转换成一个顶级的Mac程序。有关更加详细的信息请参看http://java.sun.com/developer/technicalArticles/JavaLP/JavaToMac3。

10.1.3 资源

在applet和应用程序中使用的类通常需要使用一些相关的数据文件，例如：

- 图像和声音文件。
- 带有消息字符串和按钮标签的文本文件。
- 二进制数据文件，例如，描述地图布局的文件。

在Java中，这些关联的文件被称为资源（resource）。

> **注释**：在Windows中，术语“资源”有着更加特殊的含义。Windows资源也是由图像、按钮标签等组成，但是它们都附属于可执行文件，并通过标准的程序设计访问。相比之下，Java资源作为单独的文件存储，并不是作为类文件的一部分存储。对资源的访问和解释由每个类自己胜任。

例如，AboutPanel类显示了一条信息，如图10-1所示。

当然，在面板中的书名和版权年限将会在出版下一版图书时发生变化。为了易于追踪这个变化，希望将文本放置在一个文件中，而不是以字符串的形式硬写到代码中。

但是应该将about.txt这样的文件放在哪儿呢？显然，将它与其他程序文件一起放在JAR文件中是最方便的。

类加载器知道如何搜索类文件，直到在类路径、存档文件或web服务器上找到为止。利用资源机制，对于非类文件也可以同样方便地进行操作。下面是必要的步骤：

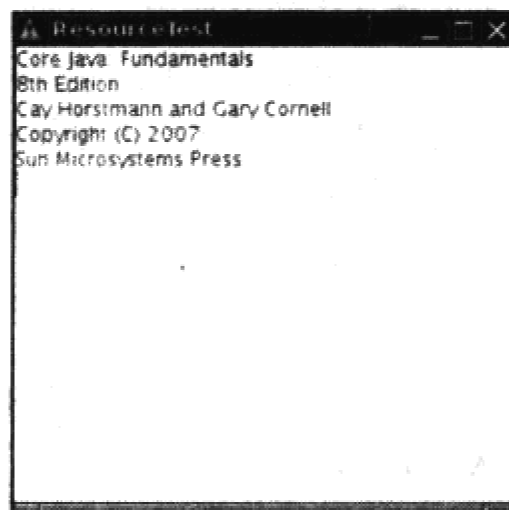


图10-1 显示来自于JAR文件的资源

1）获得具有资源的Class对象，例如，AboutPanel.class。

2）如果资源是一个图像或声音文件，那么就需要调用getresource (filename)获得作为URL的资源位置，然后利用getImage或getAudioClip方法进行读取。

3）与图像或声音文件不同，其他资源可以使用getResourceAsStream方法读取文件中的数据。

重点在于类加载器可以记住如何定位类，然后在同一位置查找关联的资源。

例如，要想利用about.gif图像文件制作图标，可以使用下列代码：

```
URL url = ResourceTest.class.getResource("about.gif");
Image img = Toolkit.getDefaultToolkit().getImage(url);
```

这段代码的含义是“在找到ResourceTest类的地方定位about.gif文件”。

要想读取about.txt文件，可以使用下列命令：

```
InputStream stream = ResourceTest.class.getResourceAsStream("about.txt");
Scanner in = new Scanner(stream);
```

除了可以将资源文件与类文件放在同一个目录中外，还可以将它放在子目录中。可以使用下面所示的层级资源名：

```
data/text/about.txt
```

这是一个相对的资源名，它会被解释为相对于加载这个资源的类所在的包。注意，必须使用“/”作为分隔符，而不要理睬存储资源文件的系统实际使用哪种目录分隔符。例如，在Windows文件系统中，资源加载器会自动地将“/”转换成“\”。

一个以“/”开头的资源名被称为绝对资源名。它的定位方式与类在包中的定位方式一样。例如，资源

```
/corejava/title.txt
```

定位于corejava目录下（它可能是类路径的一个子目录，也可能位于JAR文件中，对applet来说在web服务器上）。

文件的自动装载是利用资源加载特性完成的。没有标准的方法来解释资源文件的内容。每个程序必须拥有解释资源文件的方法。

另一个经常使用资源的地方是程序的国际化。与语言相关的字符串，如消息和用户界面标签都存放在资源文件中，每种语言对应一个文件。国际化API（internationalization API）将在卷II的第5章中进行讨论。这些API提供了组织和访问本地化文件的标准方法。

例10-1显示了这个程序的原代码。这个程序演示了资源加载。编译、创建JAR文件和执行这个程序的命令是：

```
javac ResourceTest.java
jar cvfm ResourceTest.jar ResourceTest.mf *.class *.gif *.txt
java -jar ResourceTest.jar
```

将JAR文件移到另外一个不同的目录中，再运行它，以便确认程序是从JAR文件中，而不是从当前目录中读取的资源。

例10-1 ResourceTest.java

```java
import java.awt.*;
import java.io.*;
import java.net.*;
import java.util.*;
import javax.swing.*;

/**
 * @version 1.4 2007-04-30
 * @author Cay Horstmann
 */
public class ResourceTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               ResourceTestFrame frame = new ResourceTestFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
```

```
}

/**
 * A frame that loads image and text resources.
 */
class ResourceTestFrame extends JFrame
{
   public ResourceTestFrame()
   {
      setTitle("ResourceTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
      URL aboutURL = getClass().getResource("about.gif");
      Image img = Toolkit.getDefaultToolkit().getImage(aboutURL);
      setIconImage(img);

      JTextArea textArea = new JTextArea();
      InputStream stream = getClass().getResourceAsStream("about.txt");
      Scanner in = new Scanner(stream);
      while (in.hasNext())
         textArea.append(in.nextLine() + "\n");
      add(textArea);
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 300;
}
```

API java.lang.Class 1.0

- URL getResource(String name) 1.1
- InputStream getResourceAsStream(String name) 1.1

找到与类位于同一位置的资源，返回一个可以加载资源的URL或者输入流。如果没有找到资源，则返回null，而且不会抛出异常或者发生I/O错误。

10.1.4 密封

在第4章曾经提到过，可以将Java包密封（seal）以保证不会有其他的类加入到其中。如果在代码中使用了包可见的类、方法和域，就可能希望密封包。如果不密封，其他类就有可能放在这个包中，进而访问包可见的特性。

例如，如果密封了com.mycompany.util包，就不能用下面的语句顶替密封包之外的类：

```
package com.mycompany.util;
```

要想密封一个包，需要将包中的所有类放到一个JAR文件中。在默认情况下，JAR文件中的包是没有密封的。可以在清单文件的主节中加入下面一行：

```
Sealed: true
```

来改变全局的默认设定。对于每个单独的包，可以通过在JAR文件的清单中增加一节，来指定是否想要密封这个包。例如：

```
Name: com/mycompany/util/
Sealed: true
```

```
Name: com/mycompany/misc/
Sealed: false
```

要想密封一个包，需要创建一个包含清单指令的文本文件。然后用常规的方式运行jar命令：

```
jar cvfm MyArchive.jar manifest.mf files to add
```

10.2 Java Web Start

Java Web Start是一项在Internet上发布应用程序的技术。Java Web Start 应用程序包含下列主要特性：

- Java Web Start应用程序一般通过浏览器发布。只要Java Web Start应用程序下载到本地就可以启动它，而不需要浏览器。
- Java Web Start应用程序并不在浏览器窗口内。它将显示在浏览器外的一个属于自己的框架中。
- Java Web Start应用程序不使用浏览器的Java实现。浏览器只是在加载Java Web Start 应用程序描述符时启动一个外部应用程序。这与启动诸如Adobe Acrobat 或 RealAudio这样的辅助应用程序的所使用的机制一样。
- 数字签名应用程序可以被赋予访问本地机器的任意权限。未签名的应用程序只能运行在"沙箱"中，它可以阻止具有潜在危险的操作。

要想准备一个通过Java Web Start 发布的应用程序，应该将其打包到一个或多个JAR文件中。然后创建一个Java Network Launch Protocol (JNLP)格式的描述符文件。将这些文件放置在web服务器上。

还需要确保web服务器对扩展名为.jnlp的文件报告一个application/x-java-jnlp-file 的MIME类型（浏览器利用MIME类型确定启动哪一种辅助应用程序）。有关更详细的信息请参看web服务器文档。

提示：要想体检Java Web Start，需要从jakarta.apache.org/tomcat上安装Tomcat。Tomcat是一个servlets和 JSP页面的容器，也提供网页服务。它被预配置为服务于JNLP文件所对应的MIME类型。

试着用Java Web Start发布第9章中开发的计算器应用程序。步骤如下：

1）编译Calculator.java.

2）准备一个含有下列内容的清单文件 Calculator.mf

```
Main-Class: Calculator
```

3）使用下列命令创建一个JAR 文件：

```
jar cvfm Calculator.jar Calculator.mf *.class
```

4）使用下列内容准备启动文件 Calculator.jnlp：

```
<?xml version="1.0" encoding="utf-8"?>
<jnlp spec="1.0+" codebase="http://localhost:8080/calculator/" href="Calculator.jnlp">
  <information>
    <title>Calculator Demo Application</title>
```

```
      <vendor>Cay S. Horstmann</vendor>
      <description>A Calculator</description>
      <offline-allowed/>
    </information>
    <resources>
      <j2se version="1.5.0+"/>
      <jar href="Calculator.jar"/>
    </resources>
    <application-desc/>
  </jnlp>
```

注意，版本号必须是1.5.0，而不是5.0。在Java SE 6中，标记名字时可以用java代替j2se。启动文件格式很容易理解，无需说明。要想了解全部的规范，请参看http://java.sun.com/products/javawebstart/docs/developersguide.htm。

5）如果使用Tomcat，则在Tomcat安装的根目录上创建一个目录 tomcat/webapps/calculator。创建子目录 tomcat/webapps/calculator/WEB-INF，并且将最小的web.xml文件放置在WEB-INF子目录下：

```
<?xml version="1.0" encoding="utf-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/j2ee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
      http://java.sun.com/xml/ns/j2ee/web-app_2_5.xsd">
</web-app>
```

6）将JAR文件和启动文件放置在web服务器上，以便于URL与JNLP文件中的codebase条目匹配。如果使用Tomcat，把它们放置在tomcat/webapps/calculator目录中。

7）通过检查application/x-java-jnlp-file MIME类型与javaws应用程序的关联情况来确保浏览器已经为Java Web Start进行了配置。如果安装了JDK，配置将自动完成。

8）启动Tomcat。

9）将浏览器指向JNLP文件。例如，如果使用Tomcat，将指向 http://localhost:8080/calculator/Calculator.jnlp。

10）可以看到Java Web Start的启动窗口，如图10-2所示。

11）稍后，计算器就会出现，所带的边框表明这是一个Java应用程序，如图10-3所示。

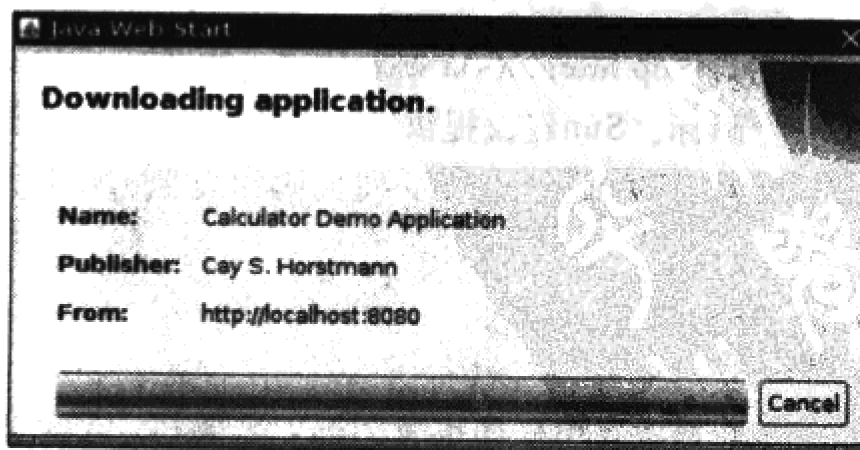


图10-2 启动Java Web Start

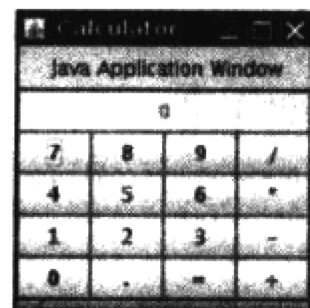


图10-3 Java Web Start发布的计算器

12）当再次访问JNLP文件时，应用程序将从缓存中取出。可以利用Java插件控制面板查看缓存内容，如图10-4所示。在JDK 5.0中，applet和Java Web Start应用程序都使用这个控制面板。在Windows中，在Windows控制面板中可以看到Java插件控制面板。在Linux下，可以运行jdk/jre/bin/ControlPanel。

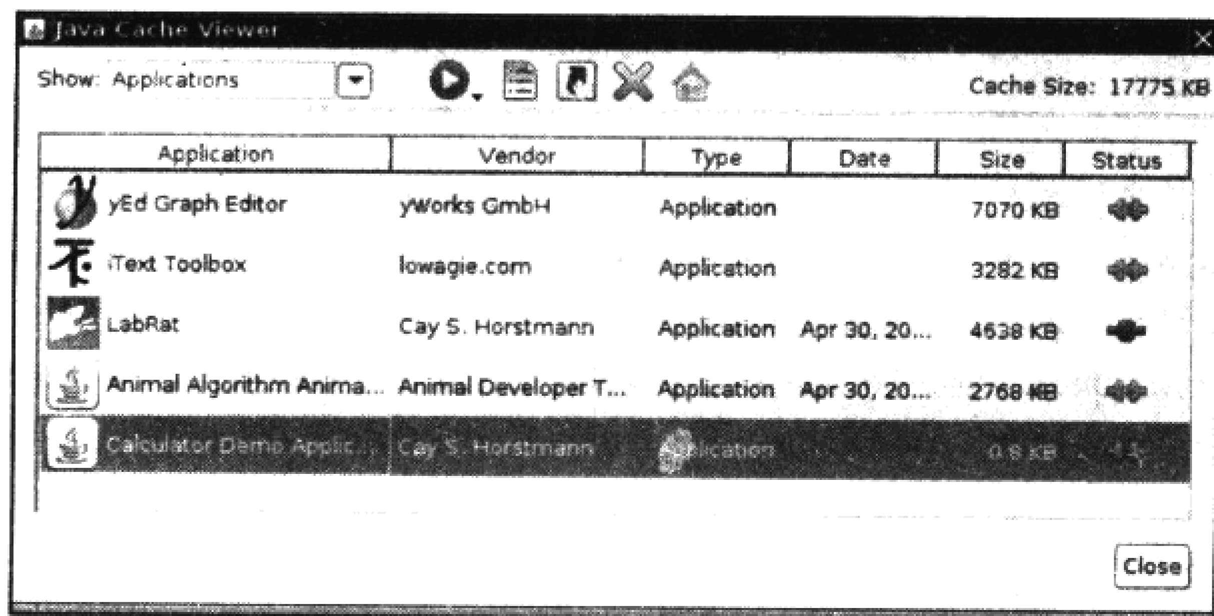


图10-4 应用程序缓存

警告：如果在测试JNLP配置期间不想运行web服务器，可以通过执行下列命令临时性地覆盖启动文件中的URL：

```
javaws -codebase file://programDirectory JNLPfile
```

例如，在UNIX中，可以从包含JNLP文件的目录中直接地发出下列命令：

```
javaws -codebase file://`pwd` WebStartCalculator.jnlp
```

当然，没有告诉用户再次运行应用程序时要启动缓存视图。可以利用安装器安装桌面和菜单的快捷键。将下列代码添加到JNLP文件中：

```
<shortcut>
   <desktop/>
   <menu submenu="Accessories"/>
</shortcut>
```

当用户首次下载应用程序时，将会显示“desktop integration warning”，如图10-5所示。

还应该为菜单快捷键和启动屏幕提供一个图标。Sun建议提供32 × 32和64 × 64的图标。把这些图标文件与JNLP和JAR文件一起放在web服务器上。将下列代码添加到JNLP文件的information节中：

```
<icon href="calc_icon32.png" width="32" height="32" />
<icon href="calc_icon64.png" width="64" height="64" />
```

注意，这些图标与应用程序图标无关。如果想让应用程序也有图标，需要将一个独立的图标图像添加到JAR文件中，并在框架类中调用thesetIconImage方法。请看示例10-1。

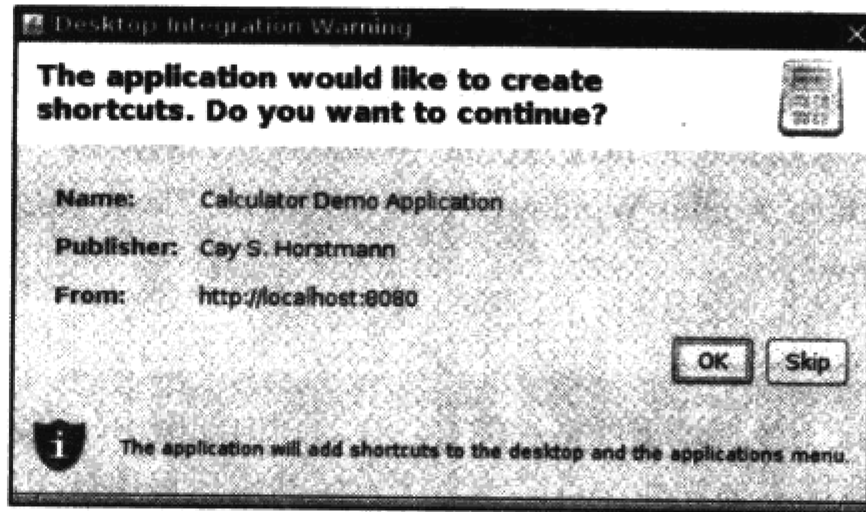


图10-5　桌面综合警告

10.2.1　沙箱

无论何时加载远程网站上代码并在本地执行，安全都是至关重要的问题。点击一个链接可以启动Java Web Start应用程序。访问一个网页时，其中的所有applet也会自动地启动。如果在点击一个链接，或者访问一个网页时，在用户的机器上能够安装任意的代码，那么犯罪分子就可能在此时窃取机密信息、读取财务数据或接管用户机器来发送广告。

为了确保Java技术不会被邪恶目的所利用，Java有一套精心设计的安全模型，具体内容在卷II中论述。*安全管理器*（security manager）将检查有权使用所有系统的资源。在默认情况下，只允许执行那些无害的操作，要想允许执行其他的操作，代码必须得到数字签名，用户必须通过数字认证。

在所有平台上，远程代码可以做什么呢？它可以显示图像、播放音乐、获得用户的键盘输入和鼠标点击，以及将用户的输入送回加载代码所在的主机。这些功能足以能够显示信息和图片，或者获得用户为订单所输入的信息。这种受限制的执行环境称为*沙箱*（sandbox）。在沙箱中运行的代码不能修改或查看用户系统。

特别是，在沙箱中的程序有下列限制：

- 不能运行任何本地的可执行程序。
- 不能从本地计算机文件系统中读取任何信息，也不能往本地计算机文件系统中写入任何信息。
- 不能查看除Java版本信息和少数几个无害的操作系统详细信息外的任何有关本地计算机的信息。特别是，在沙箱中的代码不能查看用户名、e-mail地址等信息。
- 远程加载的程序不能与除下载程序所在的服务器之外的任何主机通信，这个服务器被称为*源主机*（originating host）。这条规则通常称为“远程代码只能与家人通话。”这条规则将确保用户不会被代码探查到内部网络资源（在Java SE 6中，Java Web Start应用程序可以与其他网络连接，但必须得到程序用户的同意）。
- 所有弹出式窗口都会带一个警告消息。这条消息起到了安全的作用，以确保用户不会为本地应用程序弄错窗口。令人担心的是，一个可信赖的用户可以访问网页，并被蒙骗运行远程代码，然后输入密码或信用卡号，这些信息将被送回服务器。在早期的JDK版本

中，有个消息会令人害怕："Untrusted Java Applet Window（不可信赖的Java Applet Window）"。后来的JDK版本将这个警告消息的口气缓和了一些"Unauthenticated Java Applet Window（未获认证的Java Applet Window）"，随后是"Warning：Java Applet Window（警告：Java Applet Window）"。现在只提示"Java Applet Window"或"Java Application Window"。

10.2.2 签名代码

沙箱限制在很多情况下显得过于严格。例如，在公司内部网中，可能设想让某个Web Start应用程序或某个applet访问本地文件。可以为某个特定的应用程序赋予特定的权限来进行具体的控制。有关这些内容将在卷II的第9章中讨论。当然，应用程序可以直接请求拥有一个桌面应用程序所许可的全部权限，相当多的Java Web Start应用程序就是这样做的。只要将下列标签添加到JNLP文件中就可以达到这个目的：

```
<security>
  <all-permissions/>
</security>
```

要想在沙箱外运行，java Web Start应用程序的JAR文件必须进行数字签名。签名的JAR文件将携带一个可以证明签名者身份的证书。加密技术可以确保这种证书不可被伪造，并且检测任何企图破坏签名文件的行为。

例如，假设接到一个由yWorks GmbH开发且数字签名的应用程序，它使用了由Thawte签发的证书，如图10-6所示。当收到这个应用程序时，将要确定下列内容：

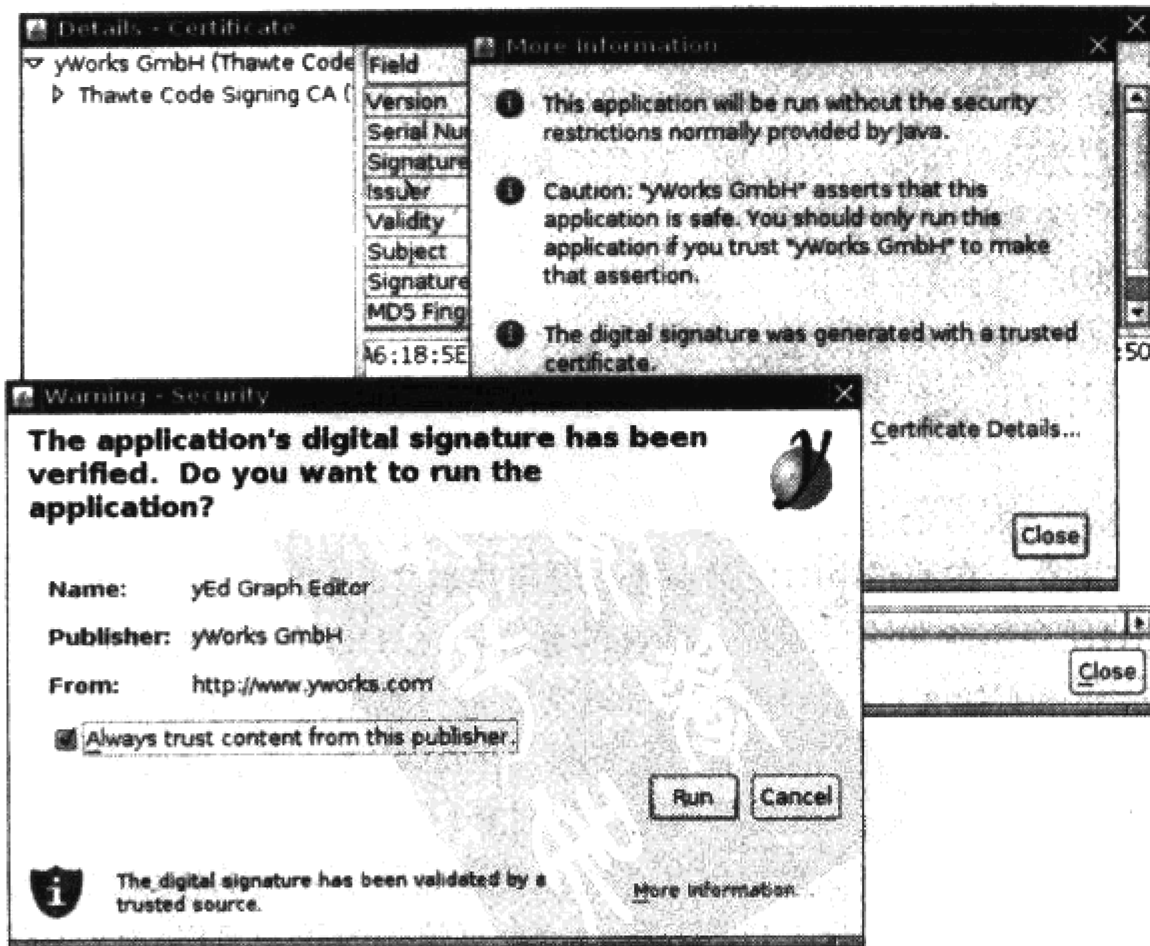


图10-6 安全认证书

1）代码确实被签名，且没有被第三方破坏。

2）由yWorks签名。

3）认证由Thawte签发（Java Web Start知道应该如何核查来自Thawte和少数其他卖方的认证）。

很不幸，这些是知道的全部内容，而并不清楚代码固有的安全性。实际上，如果点击“More Information”链接，则会被告之应用程序在没有Java提供的常规安全限制下运行。应该安装和运行应用程序吗？这将取决于yWorks GmbH对你的信任。

从一个提供证书的卖方那里获得一个证书每年需要花费数百元。很多开发者直接生成自己的证书，并利用这些证书为代码签名。当然，Java Web Start无法准确地核查这些证书。在收到这样一个应用程序时，可以知道：

1）代码确实被签名，且没有被第三方破坏。

2）有些代码被签名，但Java Web Start无法确认哪个代码已经被签名。

这没有任何价值，因为任何声称自己是作者的人都可以先破坏代码，然后再签名。然而，Java Web Start完全陶醉于得到了正式批准的证书（如图10-7所示）。从理论上讲，可以用其他方式验证证书，但是，几乎没有用户能够有这样的技术头脑。

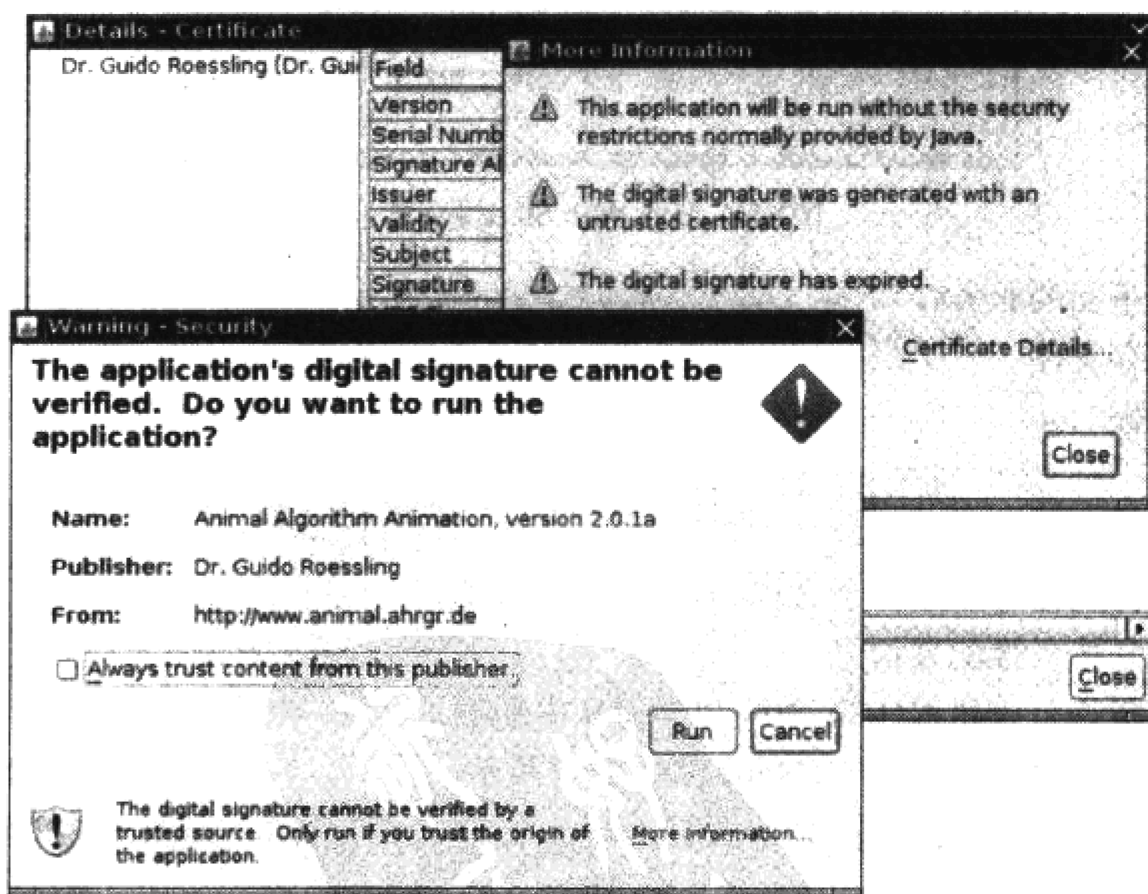


图10-7 不安全的证书

当然，很多人每天都会从Internet下载并运行应用程序。如果发现用户信任这些应用程序和web的下层链接，就可以继续使用自签名的证书（有关更加详细的内容请参看http://java.sun.com/javase/6/docs/technotes/guides/javaws/developersguide/development.html）。如果用户不信任这些应用程序，就要为用户提供安全的保证，将这些应用程序方在沙箱中。利用JNLP API（稍后将讨论）仍然允许应用程序有选择地访问资源和获得用户许可的内容。

10.2.3 JNLP API

JNLP API允许未签名的应用程序在沙箱中运行，同时通过一种安全的途径访问本地资源。例如，有一些加载和保存文件的服务。应用程序不能查看系统文件，也不能指定文件名。然而，可以弹出一个文件对话框，程序用户可以从中选择文件。在文件对话框弹出之前，应用程序不能浏览文件系统，也不能指定文件名。取而代之的是，系统将弹出文件对话框，由程序的用户从中选择文件。在文件对话框弹出之前，程序的用户会得到警告并且必须同意继续处理，如图10-8所示。而且，API并不给予程序访问File对象的权限。特别是，应用程序没有办法找到文件的位置。因此，程序员需要通过工具实现“打开文件”和“保存文件”的操作，但是对于不信任的应用程序，系统应尽可能地将信息隐藏起来。

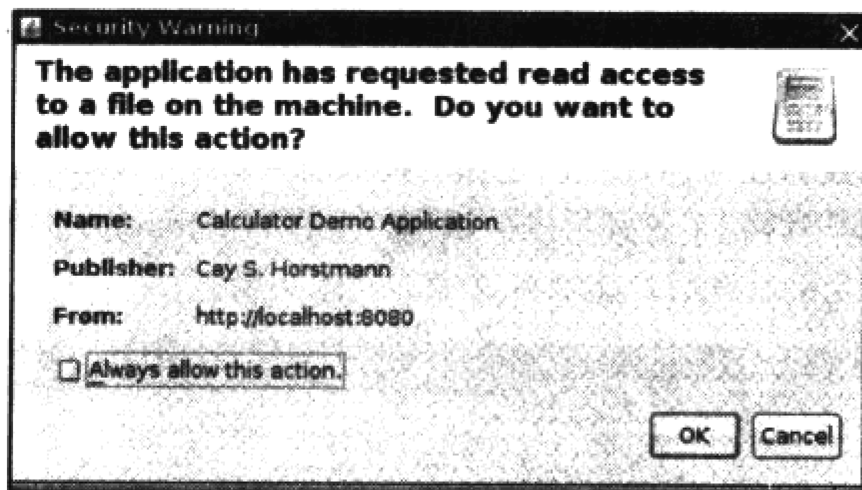


图10-8 Java Web Start安全警告

API提供了下面的服务：

- 加载和保存文件
- 访问剪贴板
- 打印
- 下载文件
- 在默认的浏览器中显示一个文档
- 保存和恢复持久性配置信息
- 确信只运行一个应用程序的实例（JDK5.0 中新增）

要访问服务，需要使用ServiceManager，如下所示：

```
FileSaveService service = (FileSaveService) ServiceManager.lookup("javax.jnlp.FileSaveService");
```

如果服务不可用，调用将抛出UnavailableServiceException。

注释：如果想要编译使用了JNLP API的程序，那就必须在类路径中包含javaws.jar文件。这个文件在JDK的jre/lib子目录下。

现在，讨论一些最常用的JNLP服务。要保存文件，需要为文件对话框提供文件的初始路径名、文件名和扩展类型。例如：

```
service.saveFileDialog(".", new String[] { "txt" }, data, "calc.txt");
```

数据必须由InputStream 传递。这里有一些小窍门。在例10-2中，程序使用下面的策略：

1）建立ByteArrayOutputStream用于存放需要保存的字节。

2）建立PrintStream用于将数据传递给ByteArrayOutputStream。

3）将要保存的数据打印到PrintStream。

4）建立ByteArrayInputStream用于读取保存的字节。

5）将上面的流传递给saveFileDialog方法。

卷II的第1章将阐述更多有关流的知识。现在，可以忽略示例中关于流的细节。

要想从文件中读取数据，需要使用FileOpenService。它的openFileDialog对话框接收应用程序提供的初始路径名和文件扩展名，并返回一个FileContents对象。然后调用getInputStream和getOuputStream方法来读写文件数据。如果用户没有选择文件， openFileDialog方法将返回null。

```
FileOpenService service = (FileOpenService) ServiceManager.lookup("javax.jnlp.FileOpenService");
FileContents contents =  service.openFileDialog(".", new String[] { "txt" });
if (contents != null)
{
   InputStream in = contents.getInputStream();
   . . .
}
```

注意，应用程序不知道文件名或文件所放置的位置。相反地，如果想要打开一个特定的文件，需要使用ExtendedService：

```
ExtendedService service = (ExtendedService) ServiceManager.lookup("javax.jnlp.ExtendedService");
FileContents contents =  service.openFile(new File("c:\\autoexec.bat"));
if (contents != null)
{
   OutputStream out = contents.getOutputStream();
   . . .
}
```

程序的用户必须同意文件访问。如图10-9所示。

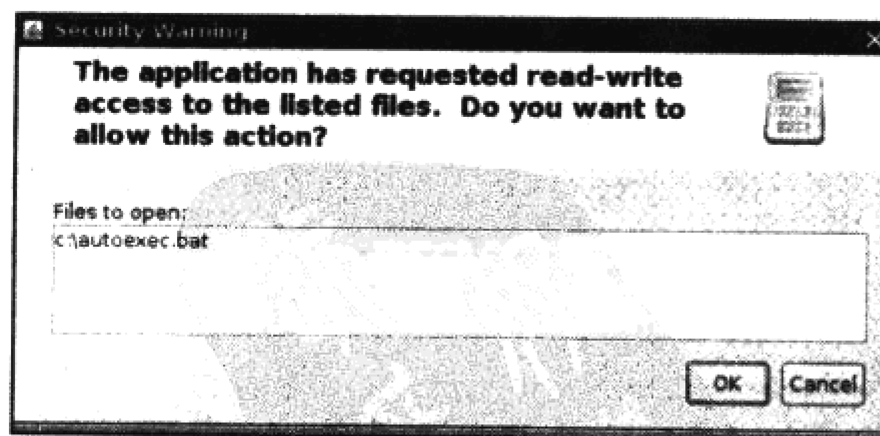


图10-9 文件访问警告

要想在默认浏览器中显示一个文档，就需要使用BasicService 接口。注意，有些系统（特别是UNIX和Linux系统）可能没有默认的浏览器。

```
BasicService service = (BasicService) ServiceManager.lookup("javax.jnlp.BasicService");
if (service.isWebBrowserSupported())
   service.showDocument(url);
```

```
else . . .
```

处于起步阶段的PersistenceService允许应用程序保存少量的配置信息，并且在应用程序下次运行时恢复。这种机制类似于HTTP的cookie，使用URL键进行持久性存储。URL并不需要指向一个真正的web资源，服务仅仅使用它们来构造一个方便的具有层次结构的名称空间。对于任何给定的URL键，应用程序可以存储任意的二进制数据（存储可能会限制数据块的大小）。

因为应用程序是彼此分离的，一个特定的应用程序只能使用从codebase开始的URL键值（codebase在JNLP文件中指定）。例如，如果一个应用程序是从http://myserver.com/apps下载的，那么它只能使用http://myserver.com/apps/subkey1/subkey2/...形式的键。访问其他键值的企图终将会失败。

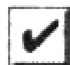
应用程序可以调用BasicService类的getCodeBase方法查看它的codebase。可以调用PersistenceService中的create方法建立一个新的键：

```
URL url = new URL(codeBase, "mykey");
service.create(url, maxSize);
```

要想访问与某个特定键关联的信息，需要调用get方法。这个方法将返回一个FileContents对象，通过这个对象可以对键数据进行读写。例如：

```
FileContents contents = service.get(url);
InputStream in = contents.getInputStream();
OutputStream out = contents.getOutputStream(true); // true = overwrite
```

遗憾的是，无法轻而易举地知道这个键是已经存在还是需要创建。可以假定这个键已经存在，并调用get方法。如果抛出FileNotFoundException，就说明需要创建。


注释：从JDK 5.0开始，Java Web Start 应用程序和applet都可以使用常规的打印API进行打印。这时会弹出一个安全对话框来询问用户是否允许访问打印机。有关更多的打印API的信息,请参看卷II的第7章。

例10-2对计算器应用程序的功能做了一点改进。这个计算器有一个虚拟的纸带来记录所有的计算过程。用户可以保存或读取计算历史信息。为了演示持久性存储功能，应用程序允许设置框架的标题。如果再次运行程序，应用程序会从持久性存储中得到标题（如图10-10所示）。

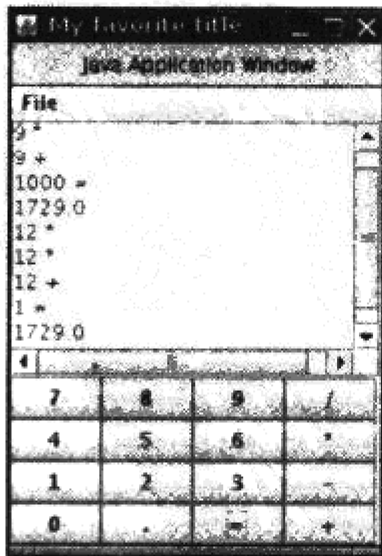


图10-10 WebStartCalculator应用程序

例10-2 WebStartCalculator.java

```
import java.awt.EventQueue;
import java.awt.event.*;
import java.io.*;
import java.net.*;
import javax.swing.*;
import javax.jnlp.*;

/**
 * A calculator with a calculation history that can be deployed as a Java Web Start application.
 * @version 1.02 2007-06-12
 * @author Cay Horstmann
 */
public class WebStartCalculator
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               CalculatorFrame frame = new CalculatorFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * A frame with a calculator panel and a menu to load and save the calculator history.
 */
class CalculatorFrame extends JFrame
{
   public CalculatorFrame()
   {
      setTitle();
      panel = new CalculatorPanel();
      add(panel);

      JMenu fileMenu = new JMenu("File");

      JMenuItem openItem = fileMenu.add("Open");
      openItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               open();
            }
         });

      JMenuItem saveItem = fileMenu.add("Save");
      saveItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
```

```
                save();
             }
          });
       JMenuBar menuBar = new JMenuBar();
       menuBar.add(fileMenu);
       setJMenuBar(menuBar);

       pack();
    }

    /**
     * Gets the title from the persistent store or asks the user for the title if there is no
     * prior entry.
     */
    public void setTitle()
    {
       try
       {
          String title = null;

          BasicService basic = (BasicService) ServiceManager.lookup("javax.jnlp.BasicService");
          URL codeBase = basic.getCodeBase();

          PersistenceService service = (PersistenceService) ServiceManager
                .lookup("javax.jnlp.PersistenceService");
          URL key = new URL(codeBase, "title");

          try
          {
             FileContents contents = service.get(key);
             InputStream in = contents.getInputStream();
             BufferedReader reader = new BufferedReader(new InputStreamReader(in));
             title = reader.readLine();
          }
          catch (FileNotFoundException e)
          {
             title = JOptionPane.showInputDialog("Please supply a frame title:");
             if (title == null) return;

             service.create(key, 100);
             FileContents contents = service.get(key);
             OutputStream out = contents.getOutputStream(true);
             PrintStream printOut = new PrintStream(out);
             printOut.print(title);
          }
          setTitle(title);
       }
       catch (UnavailableServiceException e)
       {
          JOptionPane.showMessageDialog(this, e);
       }
       catch (MalformedURLException e)
       {
          JOptionPane.showMessageDialog(this, e);
       }
       catch (IOException e)
       {
```

```
         JOptionPane.showMessageDialog(this, e);
      }
   }

   /**
    * Opens a history file and updates the display.
    */
   public void open()
   {
      try
      {
         FileOpenService service = (FileOpenService) ServiceManager
               .lookup("javax.jnlp.FileOpenService");
         FileContents contents = service.openFileDialog(".", new String[] { "txt" });

         JOptionPane.showMessageDialog(this, contents.getName());
         if (contents != null)
         {
            InputStream in = contents.getInputStream();
            BufferedReader reader = new BufferedReader(new InputStreamReader(in));
            String line;
            while ((line = reader.readLine()) != null)
            {
               panel.append(line);
               panel.append("\n");
            }
         }
      }
      catch (UnavailableServiceException e)
      {
         JOptionPane.showMessageDialog(this, e);
      }
      catch (IOException e)
      {
         JOptionPane.showMessageDialog(this, e);
      }
   }

   /**
    * Saves the calculator history to a file.
    */
   public void save()
   {
      try
      {
         ByteArrayOutputStream out = new ByteArrayOutputStream();
         PrintStream printOut = new PrintStream(out);
         printOut.print(panel.getText());
         InputStream data = new ByteArrayInputStream(out.toByteArray());
         FileSaveService service = (FileSaveService) ServiceManager
               .lookup("javax.jnlp.FileSaveService");
         service.saveFileDialog(".", new String[] { "txt" }, data, "calc.txt");
      }
      catch (UnavailableServiceException e)
      {
         JOptionPane.showMessageDialog(this, e);
      }
```

```
      catch (IOException e)
      {
         JOptionPane.showMessageDialog(this, e);
      }
   }

   private CalculatorPanel panel;
}
```

API javax.jnlp.ServiceManager

- `static String[] getServiceNames()`

  返回所有可用的服务名称。

- `static Object lookup(String name)`

  返回给定名称的服务。

API javax.jnlp.BasicService

- `URL getCodeBase()`

  返回应用程序的codebase。

- `boolean isWebBrowserSupported()`

  如果Web Start环境可以启动浏览器，返回true。

- `boolean showDocument(URL url)`

  尝试在浏览器中显示一个给定的URL。如果请求成功，返回true。

API javax.jnlp.FileContents

- `InputStream getInputStream()`

  返回一个读取文件内容的输入流。

- `OutputStream getOutputStream(boolean overwrite)`

  返回一个对文件进行写操作的输出流。如果overwrite为true，文件中已有的内容将被覆盖。

- `String getName()`

  返回文件名（但不是完整的目录路径）。

- `boolean canRead()`
- `boolean canWrite()`

  如果后台文件可以读写，返回true。

API javax.jnlp.FileOpenService

- `FileContents openFileDialog(String pathHint, String[] extensions)`
- `FileContents[] openMultiFileDialog(String pathHint, String[] extensions)`

  显示警告信息并打开文件选择器。返回文件的内容描述符或者用户选择的文件。如果用户没有选择文件，返回null。

API `javax.jnlp.FileSaveService`

- `FileContents saveFileDialog(String pathHint, String[] extensions, InputStream data, String nameHint)`
- `FileContents saveAsFileDialog(String pathHint, String[] extensions, FileContents data)`

显示警告信息并打开文件选择器。写入数据，并返回文件的内容描述符或者用户选择的文件。如果用户没有选择文件，返回null。

API `javax.jnlp.PersistenceService`

- `long create(URL key, long maxsize)`

  对于给定的键创建一个持久性存储条目。返回由持久性存储授予这个条目的最大存储空间。
- `void delete(URL key)`

  删除给定键的条目。
- `String[] getNames(URL url)`

  返回由给定的URL开始的全部键的相对键名。
- `FileContents get(URL key)`

  返回用来修改与给定键相关的数据的内容描述符。如果没有与键相关的条目，抛出FileNotFoundException。

10.3 Applet

Applet是一种包含在HTML网页中的Java应用程序。HTML网页必须告诉浏览器要加载哪个applet以及每个applet放置在页面的哪个位置。正如所预计的那样，使用applet的标记需要告诉浏览器从哪里获得类文件，以及在网页上如何定位applet（大小、位置），然后，浏览器从Internet上（或者从用户机器的目录上）获得类文件，并自动地运行applet。

在applet开发早期，必须使用Sun的HotJava 浏览器来查看包含applet的网页。自然地，很少有用户愿意专门使用一个浏览器来享受一个新的web特性。当Netscape在其浏览器中包含了Java虚拟机之后，applet才真正地流行起来。微软的IE浏览器也紧随其后。可惜的是产生了两个问题。Netscape没有跟上Java的发展，而微软在对旧版Java是否提供支持的问题上举棋不定。

为了解决这个问题，Sun发布了一个名为Java Plug-in的工具。通过使用各种扩展机制，可以将插件平滑地嵌入到各种浏览器中，使这些浏览器能够利用Sun提供的外部Java运行时环境执行Java applet。通过更新插件可以保持使用最新、最棒的Java特性。

注释：只有安装当前版本的Java Plug-in，并且保证浏览器也正确连接了Java Plug-in，才能在浏览器中运行本章介绍的applet。可以登录http://java.sun.com/getjava 寻找下载信息和配置信息。

10.3.1 一个简单的applet

根据传统习惯，首先编写一个名为NotHelloWorld 的applet程序。一个applet就是一个扩展于java.applet.Applet类的Java类。在本书中，将使用Swing实现applet.注意，所有的applet都扩展于JApplet类，它是Swing applets的超类。如图10-11所示，JApplet是Applet类的直接子类。

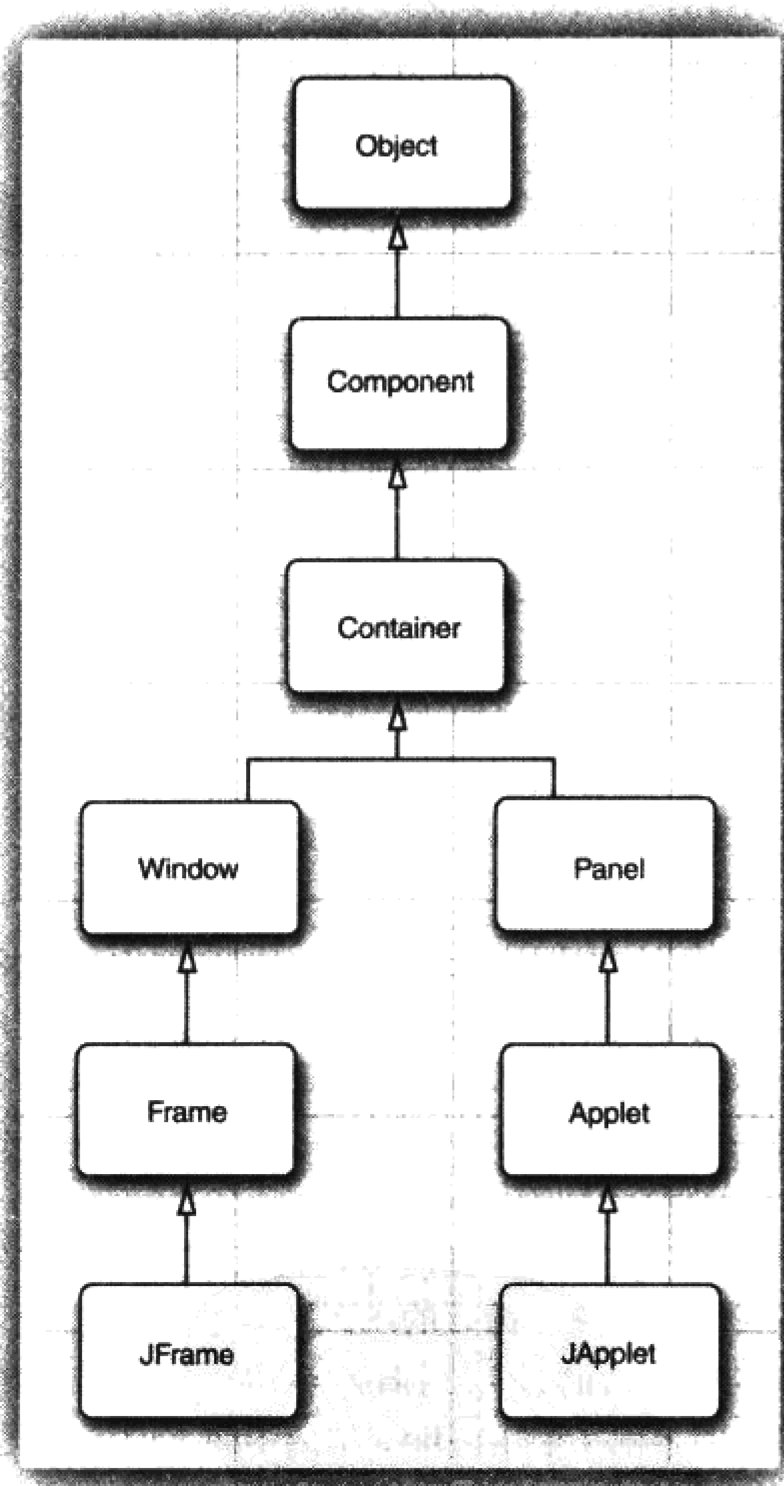


图10-11 Applet继承层次结构

注释：如果applet包含Swing组件，就必须扩展于JApplet类。在Applet类中的Swing组件不能正确地显示出来。

例10-3是“Not Hello World”的applet版本的代码。

注意，这个示例与第7章对应的程序非常相似。但是，由于applet在网页中运行，所以没有必要定义退出applet的方法。

例10-3 NotHelloWorldApplet.java

```
/*
 * The following HTML tags are required to display this applet in a browser: <applet
 * code="NotHelloWorldApplet.class" width="300" height="100"> </applet>
 */

import java.awt.*;
import javax.swing.*;

/**
 * @version 1.22 2007-06-12
 * @author Cay Horstmann
 */
public class NotHelloWorldApplet extends JApplet
{
   public void init()
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JLabel label = new JLabel("Not a Hello, World applet", SwingConstants.CENTER);
               add(label);
            }
         });
   }
}
```

想执行applet，需要执行下面两个步骤：

1）将Java源文件编译成类文件。

2）创建一个HTML文件，告诉浏览器首先加载哪个类文件以及如何设定applet的大小。

习惯上（但不是必须的），将HTML文件命名为要加载的applet类文件名。因此，根据这个习惯，这个文件名为NotHelloWorldApplet.html。下面是文件的内容：

```
<applet code="NotHelloWorldApplet.class" width="300" height="300">
</applet>
```

在浏览器中查看applet之前，最好使用JDK自带的applet查看器（applet viewer）对applet测试一下。在本示例中，要想使用applet浏览器，需要键入下列命令：

```
appletviewer NotHelloWorldApplet.html
```

Applet查看器的命令行参数是HTML文件名，而不是class文件。图10-12展示了用applet查看器显示这个applet的效果。

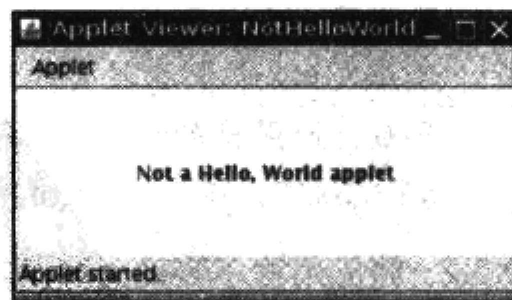


图10-12 在applet查看器中查看applet

> **!** **提示**：这里有个用来避免创建其他HTML文件的技巧，即将applet标记直接作为注释添加在源文件中：
>
> ```
> /*
> ```

```
    <applet code="MyApplet.class" width="300" height="300">
    </applet>
  */
  public class MyApplet extends JApplet
  . . .
```

然后将源文件作为命令行参数运行applet查看器：

```
appletviewer NotHelloWorldApplet.java
```

我们并不推荐使用这样方法，但是，如果想在测试阶段减少文件数目，这样方法还是很方便的。

> **提示**：也可以在集成环境中运行applet。在Eclipse中，从菜单中选择Run→Run as→Java Applet。

Applet查看器很适合用于第一阶段的测试，但有时需要在浏览器中运行applet，以用户使用的方式查看。特别是，applet查看器程序只显示applet，而没有显示周围的HTML文本。如果HTML文件包含多个applet标签，applet查看器就会弹出多个窗口。

为了正确地查看applet，应该直接地在浏览器中调用HTML文件，如图10-13所示。如果applet没有显示出来，就需要安装Java Plug-in。

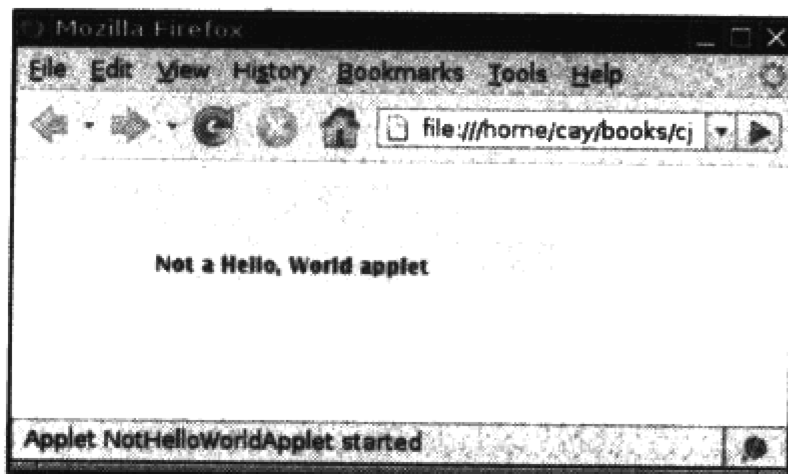


图10-13 在applet查看器中查看applet

> **提示**：如果修改了applet并重新进行了编译，就需要重新启动浏览器以便加载新的类文件。只刷新HTML页面是不会加载新代码的。在调试applet时显得非常麻烦。通过启动Java控制台并发出x命令清理类加载器缓存可以避免烦人的浏览器重启。然后可以重新加载HTML页面，使用新的applet代码。在Windows下，打开Windows控制面板中的Java Plug-in 控制。在Linux下，运行jcontrol并请求Java控制台显示。当加载applet时，控制台就会弹开。

10.3.2 将应用程序转换为applet

将一个图形的Java应用程序转换为能够嵌入在网页中运行的applet非常容易。从本质上说，所有用户界面编码都是相同的。下面是将应用程序转化为applet的基本步骤：

1）创建一个HTML页面，并用适当的标记加载applet代码。

2）创建一个JApplet类的子类。将这个子类标记为公有。否则，不能加载apple。

3）删去应用程序中的main方法。不要为应用程序构造框架窗口。应用程序将显示在浏览

器中。

4）将框架窗口构造器中的初始化代码移到applet的init方法中。不需要明确地构造applet对象，浏览器负责实例化并调用init方法。

5）删除对setSize的调用。在applet中，大小由HTML中的width和height参数确定定义。

6）删除对setDefaultCloseOperation的调用。不要关闭applet，退出浏览器时它将会终止运行。

7）如果应用程序调用setTitle，要删除这个调用。Applet没有标题栏。（当然，可以使用HTML的title标记为网页设置标题。）

8）不要调用setVisible(true)。Applet会被自动显示出来。



注释：稍后，将会介绍如何编写程序，使其既是一个applet，又是一个应用程序。



java.applet.Applet 1.0

- `void init()`

  首次加载applet时调用这个方法。覆盖这个方法，并且将所有的初始化代码放在这里。

- `void start()`

  覆盖这个方法，将用户每次访问包含applet的网页时需要执行的代码放入其中。典型的操作是重新激活线程。

- `void stop()`

  覆盖这个方法，将用户每次离开包含applet网页时需要执行的代码放入其中。典型的操作是撤销线程。

- `void destroy()`

  覆盖这个方法，将用户退出浏览器时需要执行的代码放入其中。

- `void resize(int width, int height)`

  请求调整applet的尺寸。如果这个方法能够在网页上使用就太好了。遗憾的是，由于与当前的布局机制冲突，因此，在当前的浏览器中不起作用。

10.3.3 Applet的HTML 标记和属性

下面是一个以最简单的形式使用applet标记的例子：

```
<applet code="NotHelloWorldApplet.class" width="300" height="100">
```

可以看出，code属性指出了类名，必须要包括.class扩展名，width和height属性确定容纳applet窗口的大小。两者都以像素为单位。还需要一个匹配的</applet>标记，它标志applet的HTML标记的结束。在<applet>和</applet>标记之间的的文字只有在浏览器不支持applet时才会显示出来。code、width和height属性是必需的。如果缺少任何一个，浏览器将不能加载applet。

所有这些信息应该嵌入在HTML页面中。至少，应该如下所示：

```
<html>
   <head>
      <title>NotHelloWorldApplet</title>
   </head>
   <body>
```

```
      <p>The next line of text is displayed under the auspices of Java:</p>
      <applet code="NotHelloWorldApplet.class" width="100" height="100">
        If your browser could show Java, you would see an applet here.
      </applet>
   </body>
</html>
```

在applet标记中可以使用下列属性：

- `width, height`

  这些属性是必要的，它以像素为单位，设定applet的宽度和高度。在applet查看器中，这是applet的初始大小。在applet查看器中可以调整创建的窗口大小。在浏览器中，不能调整applet的大小，因此，需要预测一下applet所需要的空间，让用户能够看到一个显示效果良好的结果。

- `align`

  这个属性指定applet的对齐方式。属性值与HTMLimg标记的align属性相同。

- `vspace, hspace`

  这些可选属性指定applet上下（vspace）及两侧（hspace）的像素值。

- `code`

  这个属性指定applet的类文件名称。这个名称不是相对于codebase（稍后介绍）的，就是在没有指定codebase时相对于当前页面的。

  路径名必须与applet类的包相匹配。例如，如果applet类在包com.mycompany中，这个属性就应该是code="com/mycompany/MyApplet.class"。也可以定义为code="com.mycompany.MyApplet.class"。但是，在这里不能使用绝对路径名。如果类文件存放在其他地方，就应该使用codebase属性。

  code属性只用于指定包含applet类的类名。当然，applet还可以包含其他类文件。当浏览器的类加载器加载包含applet的类时，它会意识到需要更多的类文件，并自动地加载这些类文件。

  code和object 属性（稍后介绍）都是必需的。

- `codebase`

  这个可选属性指出用于定位类文件的URL。可以使用绝对URL，甚至是不同的服务器。最常见的情况是，使用指向子目录的相对URL。例如，如果文件布局如下：

```
aDirectory/
     ├── MyPage.html
     └── myApplets/
              └── MyApplet.class
```

  可以在MyPage.html中使用下列标记：

```
<applet code="MyApplet.class" codebase="myApplets" width="100" height="150">
```

- `archive`

  这个可选项列出JRA文件，或包含类的文件和applet需要的其他资源。这些文件将在加载applet前就从服务器端获得而来。这种技术明显地加快了加载过程的速度。这是因为只需

要一个HTTP的请求来加载包含多个小文件的JAR文件。JAR文件之间用逗号分隔。例如：

```
<applet code="MyApplet.class"
  archive="MyClasses.jar,corejava/CoreJavaClasses.jar"
  width="100" height="150">
```

• object

这个标签用来指定包含序列化（serialized）的applet对象的文件名（当把所有的实例域写到一个文件中时，对象就被序列化了。在卷II的第1章中将讨论序列化问题）。在显示applet时，对象将从文件中反序列化，从而还原到原先的状态。当使用这个属性时，不会调用init方法，而是调用applet的start方法。在序列化applet对象之前，应该调用stop方法。这个特性对于实现持久性浏览器是非常有用的。持久性浏览器可以自动重新加载applet并且将其恢复到上一次浏览器关闭时的状态。这是一个特殊的特性，网页设计人员通常不会遇到这个特性。

在每个applet标记中，必须有code和object属性，例如：

```
<applet object="MyApplet.ser" width="100" height="150">
```

• name

脚本编写人员希望为applet指定 name属性，这样在编写脚本时，可以使用这个属性来代表相应的applet。Netscape 和 Internet Explorer都允许利用JavaScript调用页面中的applet方法。本书不是专门讲述JavaScript的书籍，所以只利用下列代码简述一下如何用JavaScript调用Java代码。

> **注释**：JavaScript是一种可以在网页内部使用的脚本语言，由Netscape开发，最初被称为LiveScript。除了语法与Java有些类似外，它基本上与Java没有什么关系。鉴于销售的目的，将它称为JavaScript。它的子集（称为ECMAScript）被标准化为ECMA-262。但是，Netscape和Microsoft在各自的浏览器中对这个标准作出了互不兼容的扩展，并没有人对此感到意外。如果想了解有关JavaScript的更多信息，我们一本书籍：《The Definitive Guide》，由O'Reilly & Associates出版，作者是David Flanagan。

要从JavaScript中访问applet，首先需要为其指定一个名字：

```
<applet code="MyApplet.class" width="100" height="150" name="mine">
</applet>
```

使用document.applets.appletname引用这个对象。例如：

```
var myApplet = document.applets.mine;
```

通过使用Netscape 和Internet Explorer提供的集成Java和JavaScript的能力，可以调用applet方法：

```
myApplet.init();
```

要使同一个页面上的两个applet之间可以直接地通信，name属性也是一个要素，必须为每个当前的applet实例指定一个名字。然后将这个名字字符串传递给AppletContext 类的getApplet方法。在本章稍后会讨论被称为applet间通信（inter-applet communication）的机制。

注释：在http://www.javaworld.com/javatips/jw-javatip80.html上，Francis Lu通过JavaScript与Java的通信解决了一个遗留长时间的问题：调整applet的大小而不再受width和height属性的限制。这是一个Java与JavaScript集成的很好示例。

- alt

  Java有可能在浏览器中被系统管理员禁用。此时可以利用alt属性显示信息。

```
<applet code="MyApplet.class" width="100" height="150"
   alt="If you activated Java, you would see my applet here">
```

如果浏览器不能处理applet，那么就会忽略不识别的applet和param标记。位于<applet>与</applet>标记之间的所有文本会由浏览器显示出来。相反，支持Java的浏览器不会显示<applet>与</applet>标记之间的文本。对于仍旧使用旧版本浏览器的用户，可以在这些标记中显示提示信息。例如：

```
<applet code="MyApplet.class" width="100" height="150">
   If your browser could show Java, you would see my applet here.
</applet>
```

10.3.4 Object标记

Object标记是HTML 4.0标准的一部分，W3 协会建议用它取代applet标记。Object标记有35个不同的属性，大部分属性（如onkeydown）只适用于编写动态HTML的用户。各种定位属性（如align和height）的用法与applet标记中所对应的属性一样。对于Java applet而言，object标记中的关键属性是classid。这个属性指定对象所在的位置。当然，object标记可以加载不同类型的对象，如：Java applets或ActiveX组件（如JavaPlug-in本身）。在codetype属性中，可以指定对象的类别。如Java applet的代码类型是application/java。下面是一个能够加载Java applet的object标记的示例：

```
<object
   codetype="application/java"
   classid="java:MyApplet.class"
   width="100" height="150">
```

注意，classid属性后面可以跟codebase属性，并与在applet标记中的用法一样。

10.3.5 使用参数向applet传递信息

与应用程序可以使用命令行信息一样，applet也可以使用嵌入在HTML中的参数。这是由使用被称为param的HTML标记连同自定义属性完成的。例如，假定想让网页决定文字在applet中的样式。可以使用下面的HTML标记：

```
<applet code="FontParamApplet.class" width="200" height="200">
   <param name="font" value="Helvetica"/>
</applet>
```

可以使用Applet类中的getParameter方法获得参数值。如下例所示：

```
public class FontParamApplet extends JApplet
{
```

```
    public void init()
    {
       String fontName = getParameter("font");
       . . .
    }
    . . .
 }
```

> **注释**：只能在applet的init方法中调用getParameter方法，而不是在构造器中调用。当applet构造器被执行时，参数还没有准备好。由于许多重要的applet布局都是由参数决定的，所以建议用户不要为applet提供构造器，而是将所有的初始化代码放到init方法中。

参数的返回值通常是字符串类型。如果需要调用数字类型的值，那就需要将字符串类型转换为数字类型。可以调用适当的方法采用标准方式进行转换，如使用Integer类中parseInt方法。

例如，如果想为字体增加一个size参数，HTML代码可以是：

```
<applet code="FontParamApplet.class" width="200" height="200">
   <param name="font" value="Helvetica"/>
   <param name="size" value="24"/>
</applet>
```

下面的源代码显示了读取整型参数的方式：

```
public class FontParamApplet extends JApplet
{
   public void init()
   {
      String fontName = getParameter("font");
      int fontSize = Integer.parseInt(getParameter("size"));
      . . .
   }
}
```

> **注释**：在param标记中定义的参数字符串应与getParameter方法中使用的字符串完全匹配。尤其是，两者都区分大小写。

除了保证参数与代码中的参数完全匹配外，还应该检查是否遗漏了size参数。通过测试字符串是否为null就可以达到这个目的。例如：

```
int fontsize;
String sizeString = getParameter("size");
if (sizeString == null) fontSize = 12;
else fontSize = Integer.parseInt(sizeString);
```

这里有一个很有用的applet，它使用了大量的参数。这个applet用于绘制如图10-14所示的柱状图。

```
<applet code="Chart.class" width="400" height="300">
   <param name="title" value="Diameters of the Planets"/>
   <param name="values" value="9"/>
   <param name="name.1" value="Mercury"/>
   <param name="name.2" value="Venus"/>
   <param name="name.3" value="Earth"/>
   <param name="name.4" value="Mars"/>
```

```
    <param name="name.5" value="Jupiter"/>
    <param name="name.6" value="Saturn"/>
    <param name="name.7" value="Uranus"/>
    <param name="name.8" value="Neptune"/>
    <param name="name.9" value="Pluto"/>
    <param name="value.1" value="3100"/>
    <param name="value.2" value="7500"/>
    <param name="value.3" value="8000"/>
    <param name="value.4" value="4200"/>
    <param name="value.5" value="88000"/>
    <param name="value.6" value="71000"/>
    <param name="value.7" value="32000"/>
    <param name="value.8" value="30600"/>
    <param name="value.9" value="1430"/>
</applet>
```

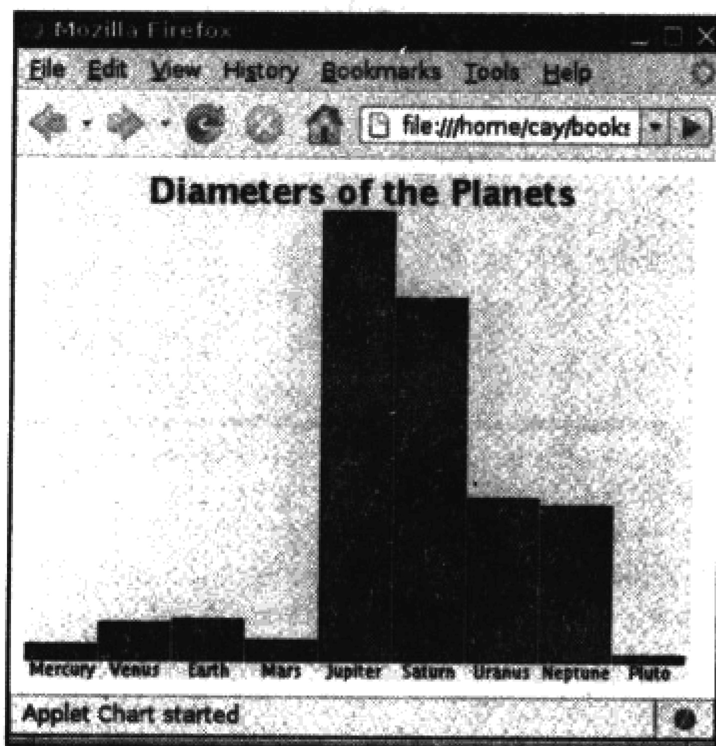


图10-14 图表applet

可以在applet中使用字符串数组和数字数组，然而使用参数机制将会有两个优势。可以在一个网页上放置同一个applet的多个拷贝以显示不同的柱状图，即只需要在页面中放置两个不同参数集的applet标记。而且还可以改变图表显示的数据。当然，像行星直径这样的数据在相当长的时间内都不会发生改变，但是，假如网页中包含的是每周销售量呢？由于使用纯文本，所以更新网页非常容易，而每周编辑和重新编译Java文件是很烦人的一件事情。

事实上，已经有很多商业JavaBeans组件（beans），它们可以显示比applet更加精美的图形。如果购买了商业JavaBean，就可以将其用到网页中，并且只需要设定参数，而不必知道applet绘制这些图形的具体方式。

例10-4是图表applet的源代码。注意，init方法读取参数，paintComponent方法绘制图表。

例10-4 Chart.java

```
import java.awt.*;
import java.awt.font.*;
import java.awt.geom.*;
```

```
import javax.swing.*;

/**
 * @version 1.33 2007-06-12
 * @author Cay Horstmann
 */
public class Chart extends JApplet
{
   public void init()
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               String v = getParameter("values");
               if (v == null) return;
               int n = Integer.parseInt(v);
               double[] values = new double[n];
               String[] names = new String[n];
               for (int i = 0; i < n; i++)
               {
                  values[i] = Double.parseDouble(getParameter("value." + (i + 1)));
                  names[i] = getParameter("name." + (i + 1));
               }

               add(new ChartComponent(values, names, getParameter("title")));
            }
         });
   }
}

/**
 * A component that draws a bar chart.
 */
class ChartComponent extends JComponent
{
   /**
    * Constructs a ChartComponent.
    * @param v the array of values for the chart
    * @param n the array of names for the values
    * @param t the title of the chart
    */
   public ChartComponent(double[] v, String[] n, String t)
   {
      values = v;
      names = n;
      title = t;
   }

   public void paintComponent(Graphics g)
   {
      Graphics2D g2 = (Graphics2D) g;

      // compute the minimum and maximum values
      if (values == null) return;
      double minValue = 0;
      double maxValue = 0;
      for (double v : values)
```

```
    {
       if (minValue > v) minValue = v;
       if (maxValue < v) maxValue = v;
    }
    if (maxValue == minValue) return;

    int panelWidth = getWidth();
    int panelHeight = getHeight();

    Font titleFont = new Font("SansSerif", Font.BOLD, 20);
    Font labelFont = new Font("SansSerif", Font.PLAIN, 10);

    // compute the extent of the title
    FontRenderContext context = g2.getFontRenderContext();
    Rectangle2D titleBounds = titleFont.getStringBounds(title, context);
    double titleWidth = titleBounds.getWidth();
    double top = titleBounds.getHeight();

    // draw the title
    double y = -titleBounds.getY(); // ascent
    double x = (panelWidth - titleWidth) / 2;
    g2.setFont(titleFont);
    g2.drawString(title, (float) x, (float) y);

    // compute the extent of the bar labels
    LineMetrics labelMetrics = labelFont.getLineMetrics("", context);
    double bottom = labelMetrics.getHeight();

    y = panelHeight - labelMetrics.getDescent();
    g2.setFont(labelFont);

    // get the scale factor and width for the bars
    double scale = (panelHeight - top - bottom) / (maxValue - minValue);
    int barWidth = panelWidth / values.length;

    // draw the bars
    for (int i = 0; i < values.length; i++)
    {
       // get the coordinates of the bar rectangle
       double x1 = i * barWidth + 1;
       double y1 = top;
       double height = values[i] * scale;
       if (values[i] >= 0) y1 += (maxValue - values[i]) * scale;
       else
       {
          y1 += maxValue * scale;
          height = -height;
       }

       // fill the bar and draw the bar outline
       Rectangle2D rect = new Rectangle2D.Double(x1, y1, barWidth - 2, height);
       g2.setPaint(Color.RED);
       g2.fill(rect);
       g2.setPaint(Color.BLACK);
       g2.draw(rect);

       // draw the centered label below the bar
       Rectangle2D labelBounds = labelFont.getStringBounds(names[i], context);
```

```
        double labelWidth = labelBounds.getWidth();
        x = x1 + (barWidth - labelWidth) / 2;
        g2.drawString(names[i], (float) x, (float) y);
      }
   }

   private double[] values;
   private String[] names;
   private String title;
}
```

java.applet.Applet 1.0

- `public String getParameter(String name)`

  获得正在加载的applet网页中由param标记定义的参数值。字符串名区分大小写。
- `public String getAppletInfo()`

  很多applet作者覆盖这个方法。它用于返回一个包含当前applet作者、版本号和版权信息的字符串。通过在applet类中覆盖这个方法，就可以创建这些信息。
- `public String[][] getParameterInfo()`

  可以覆盖这个方法，用于返回一个applet支持的param标记选项的数组。这个数组每行有三项：名字、类型和有关参数的描述。例如：

  ```
  "fps", "1-10", "frames per second"
  "repeat", "boolean", "repeat image loop?"
  "images", "url", "directory containing images"
  ```

10.3.6 访问图像和音频文件

Applet可以处理图像和音频。在编写这本书时，图像必须是GIF、PNG或JPEG 格式。音频文件必须是AU、AIFF、WAV或MIDI格式。动画支持GIF，并且能显示动画效果。

需要利用相对的URL指定图像和音频文件的位置。通常，由调用getDocumentBase或getCodeBase方法获得基URL。前者可以获得包含applet的HTML网页的URL，后者可以获得applet的codebase的URL。

注释：*在早期的Java SE版本中，这些方法相当混乱，请参看Java bug parade中的错误报告#4456393 (http://bugs.sun.com/bugdatabase/index.jsp)。这些最终在JDK 5.0中得到了澄清。*

可以通过getImage和getAudioClip方法获得基URL和文件的存储位置。例如：

```
Image cat = getImage(getCodeBase(), "images/cat.gif");
AudioClip meow = getAudioClip(getCodeBase(), "audio/meow.au");
```

第7章已经讲过如何显示图像。要想播放音频剪辑，只需要调用play方法即可。还可以调用Applet类中的play方法而不用加载音频剪辑。

```
play(getCodeBase(), "audio/meow.au");
```

API `java.applet.Applet 1.0`

- `URL getDocumentBase()`

  获得包含applet的网页的URL。
- `URL getCodeBase()`

  获得codebase目录的URL，applet是由这个目录加载的。返回的URL既可以是由codebase属性指定的引用目录的绝对URL，在没有指定codebase的情况下，也可以是HTML文件的目录。
- `void play(URL url)`
- `void play(URL url, String name)`

  前一个方法播放由URL指定的音频文件。第二个方法利用字符串提供相对于第一个参数URL的路径。如果没有找到音频剪辑，则不做任何操作。
- `AudioClip getAudioClip(URL url)`
- `AudioClip getAudioClip(URL url, String name)`

  前一个方法获得URL指定的音频剪辑。第二个方法使用字符串来提供相对于第一个参数URL的路径。如果没有找到音频剪辑，则返回null。
- `Image getImage(URL url)`
- `Image getImage(URL url, String name)`

  返回一个由URL指定的图像对象，这个对象封装一个图像。如果图像不存在，则立即返回null。否则，将启动一个独立的线程来装载图像。

10.3.7 Applet上下文

Applet在浏览器或者applet查看器中运行。Applet可以请求浏览器为它提供服务，例如，获得一段音频剪辑、在状态栏中显示简单的消息或者显示另一个网页。环境浏览器可以响应上述请求，也可以采取忽视的态度。例如，如果一个运行在applet查看器中的applet请求applet查看器显示一个网页，就不会给予响应。

如果想在浏览器之间进行通信，那么需要applet调用getAppletContext方法。这个方法将返回一个实现了AppletContext接口的对象。可以将AppletContext接口的具体实现认为是打开了一条applet与环境浏览器之间的通信道路。除了getAudioClip和getImage方法外，AppletContext接口还包含了几个很有用的方法，将在稍后介绍。

1. Applet间的通信

一个网页可以包含多个applet。如果网页中包含多个来自于同一个codebase的applet，则它们之间就可以互相通信。当然，这是一种高级技巧，也许很少用到。

如果为HTML文件中的每个applet都指定一个name属性，就可以使用AppletContext 接口中的getApplet方法来得到对applet的引用。例如，若HTML文件包含下列标记：

```
<applet code="Chart.class" width="100" height="100" name="Chart1">
```

则调用

```
Applet chart1 = getAppletContext().getApplet("Chart1");
```

就会得到对applet的引用。如何使用这个引用呢？如果Chart类包含一个利用获得的新数据重新绘制图表的方法，那么就可以在进行相应的类型转换后调用这个方法。

```
((Chart) chart1).setData(3, "Earth", 9000);
```

还可以把所有的applet都列在网页上，而不管它们是否有name属性。getApplets 方法返回一个枚举对象（enumeration object）。有关枚举更加详细的信息将在第13章中讨论。下面的循环可以打印出当前页面中包含的全部applet的类名。

```
Enumeration<Applet> e = getAppletContext().getApplets();
while (e.hasMoreElements())
{
   Applet a = e.nextElement();
   System.out.println(a.getClass().getName());
}
```

Applet不能与其他网页上的applet进行通信。

2. 在浏览器中显示信息

通过使用AppletContext 类的方法可以访问环境浏览器的两个区域：状态行和网页显示区域。可以使用showStatus消息在浏览器底部的状态行中显示一个字符串，例如：

```
showStatus("Loading data . . . please wait");
```

提示：经验证明，showStatus方法有一定的局限性。这是因为浏览器也经常使用状态行，并会用Applet running之类的话覆盖住applet在状态行上显示的有用信息，所以，应该在状态行中显示不太重要的信息，例如，“Loading data . . . please wait,”，而不要显示对用户来说非常重要的信息。

利用showDocument 方法，可以请求浏览器显示不同的网页。有几种方法可以达到这个目的。最简单的一种方法是调用只需要提供一个参数的showDocument 方法，其参数就是要显示的URL：

```
URL u = new URL("http://java.sun.com/index.html");
getAppletContext().showDocument(u);
```

这种调用方式的问题在于新打开的页面将会出现在当前网页的窗口中。因此，applet会被替换掉。要想返回applet，用户必须点击浏览器的Back按钮。

如果在调用showDocument方法时提供第二个参数，就可以告诉浏览器在另一个窗口中显示文档，请参看10-2。如果提供的特殊字符串是“_blank”，浏览器就会打开一个新窗口，而不会替换原来的窗口。更重要的是，如果利用HTML的框架特性，可以将浏览器窗口分成多个框架，每个框架都有一个名字，可以将applet放到一个框架中，并让它在另一个框架中显示文档。在下一节中，将给出这样一个示例。

注释：Sun的applet查看器不能显示网页。showDocument 方法被applet查看器忽视。

表10-2 showDocument方法

目标参数	位 置
"_self"或者none	在当前框架中显示文档
"_parent"	在父框架中显示文档
"_top"	在最顶层框架中显示文档
"_blank"	在新的、未命名的、顶层窗口中显示文档
其他字符串	在指定名称的框架中显示文档。如果没有相应名称的框架，就打开一个新窗口，并给这个窗口指定名称

3. 既是applet，又是应用程序

几年前，在一则名为“周六晚直播”的幽默电视广告中，一对夫妇正在讨论一种白色胶状物质。丈夫说：“这是一流的高级甜食。”妻子说：“这是地板蜡。”而播音员得意地宣布：“都对！”

在这一节中，将介绍如何将一个applet转换成一个Java程序，使它既是一个applet又是一个应用程序。也就是说，既可以在applet查看器和浏览器中加载这个程序，也可以通过命令行使用Java程序启动器启动这个程序。我们并不知道这种情况发生的几率，只是觉得很有趣，希望读者也是这么看的。

图10-15和图10-16显示了同一个程序。在applet查看器中运行的是applet。由命令行启动的是应用程序。

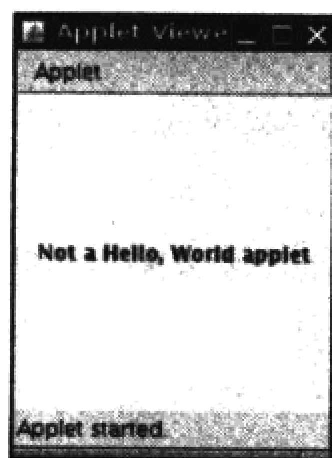


图10-15 Applet程序

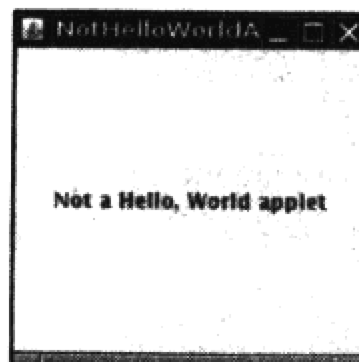


图10-16 应用程序

GUI应用程序会构造一个框架对象，并调用setVisible(true)方法。由于不能在光秃秃的applet上调用setVisible，所以必须将appet放在框架中，并在AppletFrame类提供的构造器中将applet添加到内容窗格中。

```
public class AppletFrame extends JFrame
{
   public AppletFrame(Applet anApplet)
   {
      applet = anApplet;
      add(applet);
      . . .
   }
   . . .
}
```

在applet/应用程序的main方法中，构造并显示AppletFrame。

```
class MyAppletApplication extends MyApplet
{
   public static void main(String args[])
   {
      AppletFrame frame = new AppletFrame(new MyApplet());
      frame.setTitle("MyAppletApplication");
      frame.setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}
```

注意，直接扩展于已存在的applet。

在applet启动时，必须调用它的init方法和start方法。通过覆盖AppletFrame 类中的setVisible方法可以达到这个目的：

```
public void setVisible(boolean b)
{
   if (b)
   {
      applet.init();
      super.setVisible(true);
      applet.start();
   }
   else
   {
      applet.stop();
      super.setVisible(false);
      applet.destroy();
   }
}
```

但有一个问题。如果程序以应用程序的方式启动，并且调用getAppletContext方法，就会得到一个null指针，其原因是这个程序不能在浏览器中启动。如果有下列代码，就会引发运行时错误

```
getAppletContext().showStatus(message);
```

我们并不想编写一个功能齐全的浏览器，但是需要对上述调用提供最低限度的支持。这个调用可以不显示任何信息，但至少不能导致程序崩溃。这就需要实现两个接口：AppletStub 和 AppletContext。AppletStub接口的主要用途是定位applet的上下文。每个applet都有一个applet桩（用Applet类的setStub方法设置)。下面提供了实现这两个接口的最小的基本功能。

```
public class AppletFrame extends JFrame
   implements AppletStub, AppletContext
{
   . . .
   // AppletStub methods
   public boolean isActive() { return true; }
   public URL getDocumentBase() { return null; }
   public URL getCodeBase() { return null; }
   public String getParameter(String name) { return ""; }
   public AppletContext getAppletContext() { return this; }
```

```
   public void appletResize(int width, int height) {}

   // AppletContext methods
   public AudioClip getAudioClip(URL url) { return null; }
   public Image getImage(URL url) { return Toolkit.getDefaultToolkit().getImage(url); }
   public Applet getApplet(String name) { return null; }
   public Enumeration<Applet> getApplets() { return null; }
   public void showDocument(URL url) {}
   public void showDocument(URL url, String target) {}
   public void showStatus(String status) {}
   public void setStream(String key, InputStream stream) {}
   public InputStream getStream(String key) { return null; }
   public Iterator<String> getStreamKeys() { return null; }
}
```

接下来，框架类的构造器调用applet的setStub方法，以便将applet设定为自己的桩。

```
public AppletFrame(Applet anApplet)
{
   applet = anApplet
   Container contentPane = getContentPane();
   contentPane.add(applet);
   applet.setStub(this);
}
```

使用前面提到的技巧，将applet标记以注释的方式添加到源文件中。这样，可以直接地在applet查看器中调用源程序，而不需要额外的HTML文件。

例10-5和图10-6列出了源代码。试着运行applet和应用程序：

```
appletviewer NotHelloAppletApplication.java
java NotHelloAppletApplication
```

例10-5 AppletFrame.java

```
import java.awt.*;
import java.applet.*;
import java.io.*;
import java.net.*;
import java.util.*;
import javax.swing.*;

/**
 * @version 1.32 2007-06-12
 * @author Cay Horstmann
 */
public class AppletFrame extends JFrame implements AppletStub, AppletContext
{
   public AppletFrame(Applet anApplet)
   {
      applet = anApplet;
      add(applet);
      applet.setStub(this);
   }

   public void setVisible(boolean b)
   {
      if (b)
```

```
      {
         applet.init();
         super.setVisible(true);
         applet.start();
      }
      else
      {
         applet.stop();
         super.setVisible(false);
         applet.destroy();
      }
   }

   // AppletStub methods
   public boolean isActive()
   {
      return true;
   }

   public URL getDocumentBase()
   {
      return null;
   }

   public URL getCodeBase()
   {
      return null;
   }

   public String getParameter(String name)
   {
      return "";
   }

   public AppletContext getAppletContext()
   {
      return this;
   }

   public void appletResize(int width, int height)
   {
   }

   // AppletContext methods
   public AudioClip getAudioClip(URL url)
   {
      return null;
   }

   public Image getImage(URL url)
   {
      return Toolkit.getDefaultToolkit().getImage(url);
   }

   public Applet getApplet(String name)
   {
      return null;
   }
```

```

   public Enumeration<Applet> getApplets()
   {
      return null;
   }

   public void showDocument(URL url)
   {
   }

   public void showDocument(URL url, String target)
   {
   }

   public void showStatus(String status)
   {
   }

   public void setStream(String key, InputStream stream)
   {
   }

   public InputStream getStream(String key)
   {
      return null;
   }

   public Iterator<String> getStreamKeys()
   {
      return null;
   }

   private Applet applet;
}
```

例10-6 AppletApplication.java

```
/*
 * The applet viewer reads the tags below if you call it with appletviewer AppletApplication.java
 * No separate HTML file is required. <applet code="AppletApplication.class"
 * width="200" height="200">
 * </applet>
 */

import java.awt.EventQueue;

import javax.swing.*;

/**
 * It's an applet. It's an application. It's BOTH!
 * @version 1.32 2007-04-28
 * @author Cay Horstmann
 */
public class AppletApplication extends NotHelloWorldApplet
{
   public static void main(String[] args)
   {
```

```
        EventQueue.invokeLater(new Runnable()
           {
              public void run()
              {
                 AppletFrame frame = new AppletFrame(new NotHelloWorldApplet());
                 frame.setTitle("NotHelloWorldApplet");
                 frame.setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                 frame.setVisible(true);
              }
           });
     }

     public static final int DEFAULT_WIDTH = 200;
     public static final int DEFAULT_HEIGHT = 200;
  }
```

API java.applet.Applet 1.2

- `public AppletContext getAppletContext()`

  获得applet浏览器环境的一个句柄。在大多数浏览器中，可以使用这个信息控制运行applet的浏览器。
- `void showStatus(String msg)`

  显示在浏览器状态栏中指定的字符串。

API java.applet.AppletContext 1.0

- `Enumeration <Applet> getApplets()`

  返回在同一个上下文中（也就是在同一个网页中）所有applet的枚举值（参看第13章）。
- `Applet getApplet(String name)`

  返回当前上下文中给定名称的applet。如果不存在返回null。只在当前网页中进行查找。
- `void showDocument(URL url)`
- `void showDocument(URL url, String target)`

  在浏览器的框架中显示一个新网页。第一个形式在当前页中显示新页面。第二种形式用target参数指定目标框架。请参看表10-2。

10.4 应用程序存储的配置

应用程序的用户通常期待能够自行对应用程序进行配置，并能够将其保存起来。日后再次运行这个应用程序时将能够读取这些配置。下面首先介绍一种直接将配置信息存储在属性文件中的传统方式。然后，再介绍Java SE 1.4中的一种功能更加强大的机制。

10.4.1 属性映射

属性映射（property map）是一种存储键/值对的数据结构。属性映射经常被用来存放配置信息。它有三个特性：

• 键和值都是字符串
• 键/值对可以很容易地写入文件或从文件读出。
• 用二级表存放默认值。

实现属性映射的Java类被称为Properties。

属性映射对指定程序的配置选项非常有用。例如：

```
Properties settings = new Properties();
settings.put("width", "200");
settings.put("title", "Hello, World!");
```

可以使用store方法将这个属性映射列表保存到文件中。在这里，将属性映射保存在Myprog.properties文件中。第二个参数是包含到这个文件的注释。

```
FileOutputStream out = new FileOutputStream("program.properties");
settings.store(out, "Program Properties");
```

上述示例将会输出如下结果。

```
#Program Properties
#Mon Apr 30 07:22:52  2007
width=200
title=Hello, World!
```

要想从文件中加载这些属性，可以使用：

```
FileInputStream in = new FileInputStream("program.properties");
settings.load(in);
```

习惯上，将程序属性存储在用户主目录的某个子目录下。目录名通常选择由一个（在UNIX系统中）圆点开始。这个约定表示来自用户的一个隐藏的系统目录。在示例程序中将遵守这个约定。

要想查看用户的主目录，可以调用System.getProperties方法。很巧，还可以使用Properties对象描述系统信息。主目录包含键user.home。还有一个很有用的方法，它用于读取单键：

```
String userDir = System.getProperty("user.home");
```

一旦用户手工地编辑文件，那么为应用程序提供默认值就是一种很好的想法。Properties类有两种提供默认值的机制。第一种是在试图获得字符串值时指定默认值。当键值不存在时，就会自动地使用它。

```
String title = settings.getProperty("title", "Default title");
```

如果在属性映射中有title属性，则title将设置成字符串。否则，title将设置成Default title。

如果觉得在每次调用getProperty方法时指定默认值太麻烦，那么就可以将所有的默认值放在一个二级属性映射中，并在主属性映射的构造器中提供映射。且用它来构造查询表。

```
Properties defaultSettings = new Properties();
defaultSettings.put("width", "300");
defaultSettings.put("height", "200");
defaultSettings.put("title", "Default title");
. . .
Properties settings = new Properties(defaultSettings);
```

甚至，如果为defaultSettings的构造器提供另一个属性映射参数，可以为默认值指定默认值。

但是一般不会这样做。例10-7展示了利用这些属性存储和加载程序状态的方式。程序将记住框架的位置、大小和标题。还可以手工地编辑主目录中的文件.corejava/program.properties，用以将程序的外观变成所希望的那样。

注释： 属性映射是一个没有层次结构的简单表。通过引入window.main.color、window.main.title等键名可以伪装层次结构。但是Properties类没有提供组织这种层次结构的方法。如果存储复杂的配置信息，就应该使用Preferences类，请看下一节。

例10-7 PropertiesTest.java

```
import java.awt.EventQueue;
import java.awt.event.*;
import java.io.*;
import java.util.Properties;

import javax.swing.*;

/**
 * A program to test properties. The program remembers the frame position, size, and title.
 * @version 1.00 2007-04-29
 * @author Cay Horstmann
 */
public class PropertiesTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               PropertiesFrame frame = new PropertiesFrame();
               frame.setVisible(true);
            }
         });
   }
}

/**
 * A frame that restores position and size from a properties file and updates the properties
 * upon exit.
 */
class PropertiesFrame extends JFrame
{
   public PropertiesFrame()
   {
      // get position, size, title from properties

      String userDir = System.getProperty("user.home");
      File propertiesDir = new File(userDir, ".corejava");
      if (!propertiesDir.exists()) propertiesDir.mkdir();
      propertiesFile = new File(propertiesDir, "program.properties");

      Properties defaultSettings = new Properties();
      defaultSettings.put("left", "0");
```

```java
      defaultSettings.put("top", "0");
      defaultSettings.put("width", "" + DEFAULT_WIDTH);
      defaultSettings.put("height", "" + DEFAULT_HEIGHT);
      defaultSettings.put("title", "");

      settings = new Properties(defaultSettings);

      if (propertiesFile.exists()) try
      {
         FileInputStream in = new FileInputStream(propertiesFile);
         settings.load(in);
      }
      catch (IOException ex)
      {
         ex.printStackTrace();
      }

      int left = Integer.parseInt(settings.getProperty("left"));
      int top = Integer.parseInt(settings.getProperty("top"));
      int width = Integer.parseInt(settings.getProperty("width"));
      int height = Integer.parseInt(settings.getProperty("height"));
      setBounds(left, top, width, height);

      // if no title given, ask user

      String title = settings.getProperty("title");
      if (title.equals("")) title = JOptionPane.showInputDialog("Please supply a frame title:");
      if (title == null) title = "";
      setTitle(title);

      addWindowListener(new WindowAdapter()
         {
            public void windowClosing(WindowEvent event)
            {
               settings.put("left", "" + getX());
               settings.put("top", "" + getY());
               settings.put("width", "" + getWidth());
               settings.put("height", "" + getHeight());
               settings.put("title", getTitle());
               try
               {
                  FileOutputStream out = new FileOutputStream(propertiesFile);
                  settings.store(out, "Program Properties");
               }
               catch (IOException ex)
               {
                  ex.printStackTrace();
               }
               System.exit(0);
            }
         });
   }

   private File propertiesFile;
   private Properties settings;
```

```
    public static final int DEFAULT_WIDTH = 300;
    public static final int DEFAULT_HEIGHT = 200;
}
```

API java.util.Properties 1.0

- `Properties()`

  创建一个空的属性映射。

- `Properties(Properties defaults)`

  用一组默认值创建空的属性映射。

  参数：defaults　　用于查找的默认值集合

- `String getProperty(String key)`

  获得一个属性映射。返回对应键的字符串。如果键没有在表中出现，返回在默认值表中与键对应的字符串。如果键也没有出现在默认值表中，则返回null。

  参数：key　　要获得的相关字符串的键

- `String getProperty(String key, String defaultValue)`

  如果没有找到键，返回具有默认值的属性。如果键不没有在表中出现，则返回对应键的字符串，或默认字符串。

  参数：key　　要获得的相关字符串的键

  　　　defaultValue　　如果键不存在，则返回这个字符串

- `void load(InputStream in) throws IOException`

  从输入流中加载一个属性映射。

  参数：in　　输入流

- `void store(OutputStream out, String header) 1.2`

  将属性映射存放到一个输出流中。

  参数：out　　输出流

  　　　header　　存储文件中的第一行标题

API java.lang.System 1.0

- `Properties getProperties()`

  获得全部系统属性。应用程序必须有访问全部系统属性的权限，否则将抛出安全异常。

- `String getProperty(String key)`

  返回具有给定键名的系统属性。应用程序必须有访问全部系统属性的权限，否则将抛出安全异常。下列特性无论怎样都可以得到：

```
java.version
java.vendor
java.vendor.url
java.class.version
os.name
os.version
os.arch
```

```
file.separator
path.separator
line.separator
java.specification.version
java.vm.specification.version
java.vm.specification.vendor
java.vm.specification.name
java.vm.version
java.vm.vendor
java.vm.name
```

注释：可以在Java运行时的目录下的security/java. policy文件中找到免费访问系统特性的名字。

10.4.2 Preferences API

正如读者所看到的，Properties类能够简化读取和保存配置信息的过程。但是，使用属性文件存在下列不足：

- 配置文件不能存放在用户的主目录中。这是因为某些操作系统（如Windows 9x）没有主目录的概念。
- 没有标准的为配置文件命名的规则。当用户安装了多个Java应用程序时，会增加配置文件名冲突的可能性。

有些操作系统提供一个存储配置信息的中心知识库。其中广为人知的就是Microsoft Windows。Java SE 1.4中的Preferences类提供了一个与平台无关的中心知识库。在Windows中，Preferences类用来注册存储信息；在Linux中，这些信息被存放在本地文件系统中。当然，对于使用Preferences类的程序员而言，中心知识库的实现是透明的。

Preferences的中心知识库具有树状结构，每个节点的路径类似于/com/mycompany/myapp。因为有了包名，所以只要程序员用保留的域名作为路径的开始，就可以避免命名冲突。实际上，API的设计者建议配置节点路径与程序中的包名匹配。

中心知识库中的每个节点都有一个用来存放键/值的独立表。用户可以用这张表存放数字、字符串或字节数组。但不适于存放序列化的对象。API的设计者们认为序列化格式不适宜长期存放数据。当然，如果不适宜的话，可以将序列化对象存放在字节数组中。

为了增加灵活性，系统中有多棵并行的树。每个程序使用者拥有一棵树，同时，系统中还存在一棵树，称为系统树，用于存放全体用户的公共信息。Preferences类使用操作系统中“当前用户”的概念确保用户访问恰当的用户树。

为了访问树中节点，要从用户或系统的根开始：

```
Preferences root = Preferences.userRoot();
```

或者

```
Preferences root = Preferences.systemRoot();
```

然后访问这个节点。需要直接提供节点的路径名：

```
Preferences node = root.node("/com/mycompany/myapp");
```

如果节点的路径名和类的包名一样，则有一种简单、快捷的方法获取这个节点。只需要获得这个类的对象，然后调用：

```
Preferences node = Preferences.userNodeForPackage(obj.getClass());
```

或者

```
Preferences node = Preferences.systemNodeForPackage(obj.getClass());
```

通常情况下，obj就是this引用。

一旦获得了节点，就可以使用下列方法访问键/值表。

```
String get(String key, String defval)
int getInt(String key, int defval)
long getLong(String key, long defval)
float getFloat(String key, float defval)
double getDouble(String key, double defval)
boolean getBoolean(String key, boolean defval)
byte[] getByteArray(String key, byte[] defval)
```

注意，在读取信息时，必须指定默认值，以防止中心知识库中的值不可用。有几种情况需要默认值。由于用户始终没有指定配置信息，所以数据可能找不到。在某些对资源受限的平台上可能没有中心知识库，并且移动设备可能暂时与中心知识库断开连接。

与之对应，可以利用put方法将数据写入中心知识库。

```
put(String key, String value)
putInt(String key, int value)
```

使用下列方法可以枚举出节点中全部的键：

```
String[] keys
```

但是目前无法查看给定键值的类型。

类似于Windows注册信息表这样的中心知识库通常会遇到下面两个问题。

- 其中添满了陈旧的信息，成为了“垃圾场”。
- 配置数据与中心知识库相关，使得数据不易迁移至新平台上。

Preferences类解决了第二个问题。可以调用下列方法将子树（或者一个节点）的全部值显示输出。

```
void exportSubtree(OutputStream out)
void exportNode(OutputStream out)
```

数据以XML格式存储。可以调用下列方法将它们导入到另一个中心知识库中

```
void importPreferences(InputStream in)
```

下面是一个示例文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE preferences SYSTEM "http://java.sun.com/dtd/preferences.dtd">
<preferences EXTERNAL_XML_VERSION="1.0">
  <root type="user">
    <map/>
    <node name="com">
      <map/>
      <node name="horstmann">
```

```
        <map/>
        <node name="corejava">
          <map>
            <entry key="left" value="11"/>
            <entry key="top" value="9"/>
            <entry key="width" value="453"/>
            <entry key="height" value="365"/>
            <entry key="title" value="Hello, World!"/>
          </map>
        </node>
      </node>
    </node>
  </root>
</preferences>
```

如果程序使用配置信息，那么应该允许用户导入和导出，这样就可以将这些信息从一台机器上迁移到另一台上。例10-8程序演示了这个技术。程序只保存主窗口的位置、大小和标题。然后改变窗口的大小，退出并重启应用程序。窗口的大小和位置将与关闭窗口时一样。

例10-8 PreferencesTest.java

```java
import java.awt.EventQueue;
import java.awt.event.*;
import java.io.*;
import java.util.prefs.*;
import javax.swing.*;

/**
 * A program to test preference settings. The program remembers the frame position, size,
 * and title.
 * @version 1.02 2007-06-12
 * @author Cay Horstmann
 */
public class PreferencesTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               PreferencesFrame frame = new PreferencesFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * A frame that restores position and size from user preferences and updates the preferences
 * upon exit.
 */
class PreferencesFrame extends JFrame
{
   public PreferencesFrame()
```

```
{
   // get position, size, title from preferences

   Preferences root = Preferences.userRoot();
   final Preferences node = root.node("/com/horstmann/corejava");
   int left = node.getInt("left", 0);
   int top = node.getInt("top", 0);
   int width = node.getInt("width", DEFAULT_WIDTH);
   int height = node.getInt("height", DEFAULT_HEIGHT);
   setBounds(left, top, width, height);

   // if no title given, ask user

   String title = node.get("title", "");
   if (title.equals("")) title = JOptionPane.showInputDialog("Please supply a frame title:");
   if (title == null) title = "";
   setTitle(title);

   // set up file chooser that shows XML files

   final JFileChooser chooser = new JFileChooser();
   chooser.setCurrentDirectory(new File("."));

   // accept all files ending with .xml
   chooser.setFileFilter(new javax.swing.filechooser.FileFilter()
      {
         public boolean accept(File f)
         {
            return f.getName().toLowerCase().endsWith(".xml") || f.isDirectory();
         }

         public String getDescription()
         {
            return "XML files";
         }
      });

   // set up menus
   JMenuBar menuBar = new JMenuBar();
   setJMenuBar(menuBar);
   JMenu menu = new JMenu("File");
   menuBar.add(menu);

   JMenuItem exportItem = new JMenuItem("Export preferences");
   menu.add(exportItem);
   exportItem.addActionListener(new ActionListener()
      {
         public void actionPerformed(ActionEvent event)
         {
            if (chooser.showSaveDialog(PreferencesFrame.this) == JFileChooser.APPROVE_OPTION)
            {
               try
               {
                  OutputStream out = new FileOutputStream(chooser.getSelectedFile());
                  node.exportSubtree(out);
                  out.close();
               }
```

```
                  catch (Exception e)
                  {
                     e.printStackTrace();
                  }
               }
            }
         });

      JMenuItem importItem = new JMenuItem("Import preferences");
      menu.add(importItem);
      importItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               if (chooser.showOpenDialog(PreferencesFrame.this) == JFileChooser.APPROVE_OPTION)
               {
                  try
                  {
                     InputStream in = new FileInputStream(chooser.getSelectedFile());
                     Preferences.importPreferences(in);
                     in.close();
                  }
                  catch (Exception e)
                  {
                     e.printStackTrace();
                  }
               }
            }
         });

      JMenuItem exitItem = new JMenuItem("Exit");
      menu.add(exitItem);
      exitItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               node.putInt("left", getX());
               node.putInt("top", getY());
               node.putInt("width", getWidth());
               node.putInt("height", getHeight());
               node.put("title", getTitle());
               System.exit(0);
            }
         });
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;
}
```

API `java.util.prefs.Preferences 1.4`

- `Preferences userRoot()`

  返回调用应用程序的用户配置信息的根节点。

- Preferences systemRoot()

  返回系统范围的配置信息的根节点。

- Preferences node(String path)

  返回从当前节点经过给定路径可以到达的节点。如果path是绝对路径（以 / 开始），则从包含配置信息的树根开始查找这个节点。如果给定的路径不存在这样的节点，则创建它。

- Preferences userNodeForPackage(Class cl)
- Preferences systemNodeForPackage(Class cl)

  返回当前用户树或者系统树中的节点。这个节点的绝对路径符合类cl的包名。

- String[] keys()

  返回属于这个节点的所有键。

- String get(String key, String defval)
- int getInt(String key, int defval)
- long getLong(String key, long defval)
- float getFloat(String key, float defval)
- double getDouble(String key, double defval)
- boolean getBoolean(String key, boolean defval)
- byte[] getByteArray(String key, byte[] defval)

  返回与给定键关联的值。如果没有与之关联的值，或者关联值的类型不符合要求，或者配置存储不可用，则返回默认值。

- void put(String key, String value)
- void putInt(String key, int value)
- void putLong(String key, long value)
- void putFloat(String key, float value)
- void putDouble(String key, double value)
- void putBoolean(String key, boolean value)
- void putByteArray(String key, byte[] value)

  将键/值存放在节点中。

- void exportSubtree(OutputStream out)

  将这个节点及子节点的配置信息写到指定流中。

- void exportNode(OutputStream out)

  将这个节点（不包括子节点）配置信息写到指定流中。

- void importPreferences(InputStream in)

  导入指定流中的配置信息。

至此，结束了对Java软件部署的讨论。下一章将介绍如何利用异常告诉程序在出现运行问题时如何进行处理。另外，还将介绍一些测试和调试的技巧，以保证程序运行时不会出现过多的错误。

第11章 异常、日志、断言和调试

▲ 处理异常
▲ 捕获异常
▲ 使用异常机制的技巧
▲ 使用断言
▲ 日志
▲ 调试技巧
▲ 使用调试器

在理想状态下，用户输入数据的格式永远都是正确的，选择打开的文件也一定存在，并且永远不会出现bug。迄今为止，本书呈现给大家的代码似乎都处在这样一个理想境界中。然而，在现实世界中却充满了不良的数据和带有问题的代码，现在是讨论Java程序设计语言处理这些问题的机制的时候了。

人们在遇到错误时会感觉不爽。如果一个用户在运行程序期间，由于程序的错误或一些外部环境的影响造成用户数据的丢失，用户就有可能不再使用这个程序了。为了避免这类事情的发生，至少应该做到以下几点：

- 向用户通告错误；
- 保存所有的操作结果；
- 允许用户以适当的形式退出程序。

对于异常情况，例如，可能造成程序崩溃的错误输入，Java使用一种称为*异常处理*（exception handing）的错误捕获机制处理。Java中的异常处理与C++或Delphi中的异常处理十分类似。本章的第1部分先介绍Java的异常。

在测试期间，需要进行大量的检测以验证程序操作的正确性。然而，这些检测可能非常耗时，在测试完成后也不必保留它们，因此，可以将这些检测删掉，并在其他测试需要时将它们粘贴回来，这是一件很乏味的事情。本章的第2部分将介绍如何使用断言来选择检测行为。

当程序出现错误时，并不总是能够与用户或终端进行沟通。此时，可能希望记录下出现的问题，以备日后进行分析。本章的第3部分将讨论Java SE的日志工具。

最后，介绍如何获得一个正在运行的Java应用程序的有用信息，以及如何使用IDE中的调试器的技巧。

11.1 处理异常

假设在一个Java程序运行期间出现了一个错误。这个错误可能是由于文件包含了错误信息，或者网络连接出现问题造成的，也有可能是因为使用无效的数组下标，或者试图使用一个没有被赋值的对象引用而造成的。用户期望在出现错误时，程序能够采用一些理智的行为。如果由于出现错误而使得某些操作没有完成，程序应该：

• 返回到一种安全状态，并能够让用户执行一些其他的命令；或者

• 允许用户保存所有操作的结果，并以适当的方式终止程序。

要做到这些并不是一件很容易的事情。其原因是检测（或引发）错误条件的代码通常离那些能够让数据恢复到安全状态，或者能够保存用户的操作结果，并正常地退出程序的代码很远。异常处理的任务就是将控制权从错误产生的地方转移给能够处理这种情况的错误处理器。为了能够在程序中处理异常情况，必须研究程序中可能会出现的错误和问题，以及哪类问题需要关注。

1. 用户输入错误

除了那些不可避免的打字录入外，有些用户喜欢各行其是，不遵守程序的要求。例如，假设有一个用户请求连接一个URL，而语法却不正确。在程序代码中应该对此进行检查，如果没有检查，网络数据包就会给出警告。

2. 设备错误

硬件并不总是让它做什么，它就做什么。打印机可能被关掉了。网页可能临时性地不能浏览。在一个任务的处理过程中，硬件经常出现问题。例如，打印机在打印过程中可能没有纸了。

3. 物理限制

磁盘满了，可用存储空间已被用完。

4. 代码错误

程序方法有可能无法正确的执行。例如，方法可能返回了一个错误的答案，或者错误地调用了其他的方法。使用了一个无效的数组下标，试图查找一个在散列表中不存在的数据项以及试图对一个空栈进行退栈操作。

对于方法中出现的错误，传统的处理方式是返回一个特定的错误编码，调用这个方法的方法对其进行分析。例如，对于一个从文件中读取信息的方法来说，如果返回值不是标准字符，而是一个 -1，则表示文件结束。这种处理方式对于很多异常状况都是可行的。还有一种表示错误状况的常用返回值是null引用。在第10章中，将可以看到一个使用Applet类中的getParameter方法的示例。当希望查询的参数不存在时，这个方法就会返回null。

遗憾的是，并不是在任何情况下都能够返回一个错误编码。有可能无法明确地将有效数据与无效数据加以区分。一个返回整型的方法就不能简单地通过返回 -1表示错误，因为 -1很可能是一个完全合法的结果。

正如第5章中所叙述的那样，在Java中，如果某个方法不能够采用正常的途径完整它的任务，就可以通过另外一个路径退出方法。在这种情况下，方法并不返回任何值，而是*抛出*（throw）一个封装了错误信息的对象。需要注意的是，这个方法将会立刻退出，并不返回任何值。此外，调用这个方法的代码也将无法继续执行，而是，异常处理机制开始搜索能够处理这种异常状况的*异常处理器*（exception handler）。

异常具有自己的语法和特定的继承结构。下面首先介绍一下语法，然后再给出有效地使用这种语言功能的技巧。

11.1.1 异常分类

在Java程序设计语言中，异常对象都是派生于Throwable类的一个实例。稍后还可以看到，如果Java中内置的异常类不能够满足需求，用户可以创建自己的异常类。

图11-1是Java异常层次结构的一个简化示意图。

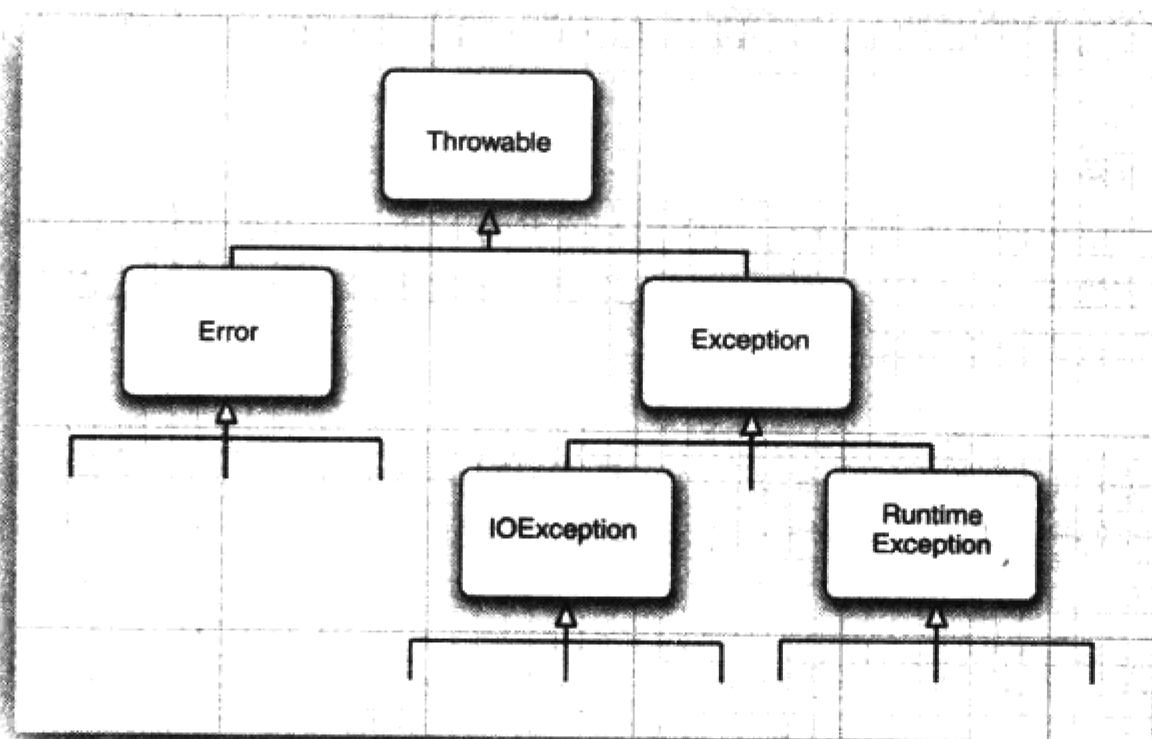


图11-1 Java中的异常层次结构

需要注意的是，所有的异常都是由Throwable继承而来，但在下一层立即分解为两个分支：Error和Exception。

Error类层次结构描述了Java运行时系统的内部错误和资源耗尽错误。应用程序不应该抛出这种类型的对象。如果出现了这样的内部错误，除了通告给用户，并尽力使程序安全地终止之外，再也无能为力了。这种情况很少出现。

在设计Java程序时，需要关注Exception层次结构。这个层次结构又分解为两个分支：一个分支派生于RuntimeException；另一个分支包含其他异常。划分两个分支的规则是：由程序错误导致的异常属于RuntimeException；而程序本身没有问题，但由于像I/O错误这类问题导致的异常属于其他异常。

派生于RuntimeException的异常包含下面几种情况：

- 错误的类型转换。
- 数组访问越界。
- 访问空指针。

不是派生于RuntimeException的异常包括

- 试图在文件尾部后面读取数据。
- 试图打开一个错误格式的URL。
- 试图根据给定的字符串查找Class对象，而这个字符串表示的类并不存在。

"如果出现RuntimeException异常，那么就一定是你的问题"是一条相当有道理的规则。应

该通过检测数组下标是否越界来避免ArrayIndexOutOfBoundsException异常；应该通过在使用变量之前检测是否为空来杜绝NullPointerException异常的发生。

如何处理错误格式的URL呢？在使用URL之前，是否也需要尽可能地判断是否“具有错误格式”呢？事实上，不同的浏览器可以处理不同类别的URL。例如，Netscape可以处理mailto:URL格式，而applet查看器就不能处理这种格式。因此，“具有错误格式”取决于具体的环境，而不仅仅是程序代码。

Java语言规范将派生于Error类或RuntimeException类的所有异常称为*未检查*（unchecked）异常，所有其他的异常称为*已检查*（checked）异常。这是两个很有用的术语，在后面还会用到。编译器将核查是否为所有的已检查异常提供了异常处理器。

注释：RuntimeException这个名字很容易让人混淆。实际上，现在讨论的所有错误都发生在运行时刻。

C++注释：如果熟悉标准C++类库中的异常层次结构，就一定会感到有些困惑。C++有两个基本的异常类，一个是runtime_error；另一个是logic_error。logic_error类相当于Java中的RuntimeException，它表示程序中的逻辑错误；runtime_error类是所有由于不可预测的原因所引发的异常的基类。它相当于Java中的非RuntimeException异常。

11.1.2 声明已检查异常

如果遇到了无法处理的情况，那么Java的方法可以抛出一个异常。这个道理很简单：一个方法不仅需要告诉编译器将要返回什么值，还要告诉编译器有可能发生什么错误。例如，一段读取文件的代码知道有可能读取的文件不存在，或者内容为空，因此，试图处理文件信息的代码就需要通告编译器可能会抛出IOException类的异常。

方法应该在其首部声明所有可能抛出的异常。这样可以从首部反映出这个方法可能抛出哪类已检查异常。例如，下面是标准类库中提供的FileInputStream类的一个构造器的声明（有关流的更多信息请参看第12章。）

```
public FileInputStream(String name) throws FileNotFoundException
```

这个声明表示这个构造器将根据给定的String参数产生一个FileInputStream对象，但也有可能抛出一个FileNotFoundException异常。如果发生了这种糟糕情况，构造器将不会初始化一个新的FileInputStream对象，而是抛出一个FileNotFoundException类对象。如果这个方法真的抛出了这样一个异常对象，运行时系统就会开始搜索异常处理器，以便知道如何处理FileNotFoundException对象。

在自己编写方法时，不必将所有可能抛出的异常都进行声明。至于什么时候需要在方法中用throws子句声明异常，什么异常必须使用throws子句声明，需要记住在遇到下面4种情况时应该抛出异常：

1）调用一个抛出已检查异常的方法，例如，FileInputStream构造器。

2）程序运行过程中发现错误，并且利用throw语句抛出一个已检查异常（下一节将详细地介绍throw语句）。

3）程序出现错误，例如，a[－1]=0会抛出一个ArrayIndexOutOfBoundsException这样的未检查异常。

4）Java虚拟机和运行时库出现的内部异常。

如果出现前两种情况之一，则必须告诉调用这个方法的程序员有可能抛出异常。因为任何一个抛出异常的方法都有可能是一个死亡陷阱。如果没有处理器捕获这个异常，当前执行的线程就会结束。

对于那些可能被他人使用的Java方法，应该根据*异常规范*（exception specification），在方法的首部声明这个方法可能抛出的异常。

```
class MyAnimation
{
   . . .
   public Image loadImage(String s) throws IOException
   {
      . . .
   }
}
```

如果一个方法有可能抛出多个已检查异常，那么就必须在方法的首部列出所有的异常类。每个异常类之间用逗号隔开。如下面这个例子所示：

```
class MyAnimation
{
   . . .
   public Image loadImage(String s) throws EOFException, MalformedURLException
   {
      . . .
   }
}
```

但是，不需要声明Java的内部错误，即从Error继承的错误。任何程序代码都具有抛出那些异常的潜能，而我们对其没有任何控制能力。

同样，也不应该声明从RuntimeException继承的那些未检查异常。

```
class MyAnimation
{
   . . .
   void drawImage(int i) throws ArrayIndexOutOfBoundsException // bad style
   {
      . . .
   }
}
```

这些运行时错误完全在我们的控制之下。如果特别关注数组下标引发的错误，就应该将更多的时间花费在修正程序中的错误上，而不是说明这些错误发生的可能性上。

总之，一个方法必须声明所有可能抛出的*已检查异常*，而未检查异常要么不可控制（Error），要么就应该避免发生（RuntimeException）。如果方法没有声明所有可能发生的已检查异常，编译器就会给出一个错误消息。

当然，从前面的示例中可以知道：除了声明异常之外，还可以捕获异常。这样会使异常不被抛到方法之外，也不需要throws规范。稍后，将会讨论如何决定一个异常是被捕获，还是被

抛出让其他的处理器进行处理。

警告：如果在子类中覆盖了超类的一个方法，子类方法中声明的已检查异常不能超过超类方法中声明的异常范围（也就是说，子类方法中抛出的异常范围更加小，或者根本不抛出任何异常）。特别需要说明的是，如果超类方法没有抛出任何已检查异常，子类也不能抛出任何已检查异常。例如，如果覆盖JComponent.paintComponent方法，由于超类中这个方法没有抛出任何异常，所以，自定义的paintComponent也不能抛出任何已检查异常。

如果类中的一个方法声明将会抛出一个异常，而这个异常是某个特定类的实例时，则这个方法就有可能抛出一个这个类的异常，或者这个类的任意一个子类的异常。例如，FileInputStream构造器声明将有可能抛出一个IOExcetion异常，然而并不知道具体是哪种IOException异常。它既可能是IOException异常，也可能是其子类的异常，例如，FileNotFoundException。

C++注释：Java中的throws说明符与C++中的throw说明符基本类似，但有一点重要的区别。在C++中，throw说明符在运行时执行，而不是在编译时执行。也就是说，C++编译器将不处理任何异常说明符。但是，如果函数抛出的异常没有出现在throw列表中，就会调用unexpected函数，这个函数的默认处理方式是终止程序的执行。

另外，在C++中，如果没有给出throw说明，函数可能会抛出任何异常。而在Java中，没有throws说明符的方法将不能抛出任何已检查异常。

11.1.3 如何抛出异常

假设在程序代码中发生了一些很糟糕的事情。一个名为readData的方法正在读取一个首部具有下列信息的文件：

```
Content-length: 1024
```

然而，读到733个字符之后文件就结束了。我们认为这是一种不正常的情况，希望抛出一个异常。

首先要决定应该抛出什么类型的异常。将上述异常归结为IOException是一种很好的选择。仔细地阅读Java API文档之后会发现：EOFException异常描述的是“在输入过程中，遇到了一个未预期的EOF后的信号”。这正是我们要抛出的异常。下面是抛出这个异常的语句：

```
throw new EOFException();
```

或者

```
EOFException e = new EOFException();
throw e;
```

下面将这些代码放在一起：

```
String readData(Scanner in) throws EOFException
{
   . . .
   while (. . .)
   {
      if (!in.hasNext()) // EOF encountered
```

```
        {
            if (n < len)
                throw new EOFException();
        }
        . . .
    }
    return s;
}
```

EOFException类还有一个含有一个字符串型参数的构造器。这个构造器可以更加细致的描述异常出现的情况。

```
String gripe = "Content-length: " + len + ", Received: " + n;
throw new EOFException(gripe);
```

在前面已经看到，对于一个已经存在的异常类，将其抛出非常容易。在这种情况下：

1）找到一个合适的异常类。

2）创建这个类的一个对象。

3）将对象抛出。

一旦方法抛出了异常，这个方法就不可能返回到调用者。也就是说，不必为返回的默认值或错误代码担忧。

C++注释：在C++与Java中，抛出异常的过程基本相同，只有一点微小的差别。在Java中，只能抛出Throwable子类的对象，而在C++中，却可以抛出任何类型的值。

11.1.4 创建异常类

在程序中，可能会遇到任何标准异常类都没有能够充分地描述清楚的问题。在这种情况下，创建自己的异常类就是一件顺理成章的事情了。我们需要做的只是定义一个派生于Exception的类，或者派生于Exception子类的类。例如，定义一个派生于IOException的类。习惯上，定义的类应该包含两个构造器，一个是默认的构造器；另一个是带有详细描述信息的构造器（超类Throwable的toString方法将会打印出这些详细信息，这在调试中非常有用）。

```
class FileFormatException extends IOException
{
    public FileFormatException() {}
    public FileFormatException(String gripe)
    {
        super(gripe);
    }
}
```

现在，就可以抛出自己定义的异常类型了。

```
String readData(BufferedReader in) throws FileFormatException
{
    . . .
    while (. . .)
    {
        if (ch == -1) // EOF encountered
        {
            if (n < len)
```

```
            throw new FileFormatException();
        }
        . . .
    }
    return s;
}
```

API java.lang.Throwable 1.0

- `Throwable()`

  构造一个新的Throwable对象，这个对象没有详细的描述信息。
- `Throwable(String message)`

  构造一个新的throwable对象，这个对象带有特定的详细描述信息。习惯上，所有派生的异常类都支持一个默认的构造器和一个带有详细描述信息的构造器。
- `String getMessage()`

  获得Throwable对象的详细描述信息。

11.2 捕获异常

到目前为止，已经知道如何抛出一个异常。这个过程十分容易。只要将其抛出就不用理睬了。当然，有些代码必须捕获异常。捕获异常需要进行周密的计划。

如果某个异常发生的时候没有在任何地方进行捕获，那程序就会终止执行，并在控制台上打印出异常信息，其中包括异常的类型和堆栈的内容。对于图形界面程序（applet和application应用程序），在捕获异常之后，也会打印出堆栈的信息，但程序将返回到用户界面的处理循环中（在调试基于图形界面的程序时，最好保证控制台窗口可见，并且没有被极小化）。

要想捕获一个异常，必须设置try/catch语句块。最简单的try语句块如下所示：

```
try
{
   code
   more code
   more code
}
catch (ExceptionType e)
{
   handler for this type
}
```

如果在try语句块中的任何代码抛出了一个在catch子句中说明的异常类，那么

1）程序将跳过try语句块的其余代码。

2）程序将执行catch子句中的处理器代码。

如果在try语句块中的代码没有抛出任何异常，那么程序将跳过catch子句。

如果方法中的任何代码抛出了一个在catch子句中没有声明的异常类型，那么这个方法就会立刻退出（期待调用者为这种类型的异常设计了catch子句）。

为了演示捕获异常的处理过程，下面给出一个读取文本的典型程序代码：

```
public void read(String filename)
```

```
{
   try
   {
      InputStream in = new FileInputStream(filename);
      int b;
      while ((b = in.read()) != -1)
      {
         process input
      }
   }
   catch (IOException exception)
   {
      exception.printStackTrace();
   }
}
```

需要注意，try语句中的大多数代码都很容易理解：读取并处理文本行，直到遇到文件结束符为止。正如在Java API中看到的那样，read方法有可能抛出一个IOException异常。在这种情况下，将跳出整个while循环，进入catch子句，并输出堆栈情况。对于一个普通的程序来说，这样处理异常基本上合乎情理。还有其他的选择吗？

通常，最好的选择是什么也不做，而是将异常传递给调用者。如果read方法出现了错误，就让read方法的调用者去操心！如果采用这种处理方式，就必须声明这个方法可能会抛出一个IOException。

```
public void read(String filename) throws IOException
{
   InputStream in = new FileInputStream(filename);
   int b;
   while ((b = in.read()) != -1)
   {
      process input
   }
}
```

请记住，编译器严格地执行throws说明符。如果调用了一个抛出已检查异常的方法，就必须对它进行处理，或者将它传递出去。

哪种方法更好呢？通常，应该捕获那些知道如何处理的异常，而将那些不知道怎样处理的异常传递出去。如果想将异常传递出去，就必须在方法的首部添加一个throws说明符，以便告知调用者这个方法可能会抛出异常。

仔细阅读一下Java API文档，以便知道每个方法可能会抛出哪种异常，然后再决定是自己处理，还是添加到throws列表中。对于后一种情况，也不必犹豫。将异常直接交给能够胜任的处理器进行处理要比压制对它的处理更好。

同时请记住，这个规则也有一个例外。前面曾经提到过：如果编写一个覆盖超类的方法，而这个方法又没有抛出异常（例如，JComponent 中的paintComponent），那么这个方法就必须捕获方法代码中出现的每一个已检查异常。不允许在子类的throws说明符中出现超过超类方法所列出的异常类范围。


C++注释：在Java与C++中，捕获异常的方式基本相同。严格地说，下列代码

```
catch (Exception e) // Java
```

与

```
catch (Exception& e) // C++
```

是一样的。

在Java中，没有与C++中catch() 对应的东西。由于Java中的所有异常类都派生于一个公共的超类，所以，没有必要使用这种机制。

11.2.1 捕获多个异常

在一个try语句块中可以捕获多个异常类型，并对不同类型的异常做出不同的处理。可以按照下列方式为每个异常类型使用一个单独的catch子句：

```
try
{
   code that might throw exceptions
}
catch (MalformedURLException e1)
{
   emergency action for malformed URLs
}
catch (UnknownHostException e2)
{
   emergency action for unknown hosts
}
catch (IOException e3)
{
   emergency action for all other I/O problems
}
```

异常对象（e1,e2,e3）可能包含与异常本身有关的信息。要想获得对象的更多信息，可以试着使用

```
e3.getMessage()
```

得到详细的错误信息（如果有的话），或者使用

```
e3.getClass().getName()
```

得到异常对象的实际类型。

11.2.2 再次抛出异常与异常链

在catch子句中可以抛出一个异常，这样做的目的是改变异常的类型。如果开发了一个供其他程序员使用的子系统，那么，用于表示子系统故障的异常类型可能会产生多种解释。ServletException就是这样一个异常类型的例子。执行servlet的代码可能不想知道发生错误的细节原因，但希望明确地知道servlet是否有故障。

下面给出了捕获异常并将它再次抛出的基本方法：

```
try
{
   access the database
}
```

```
catch (SQLException e)
{
   throw new ServletException("database error: " + e.getMessage());
}
```

这里，ServleException用带有异常信息文本的构造器来构造。在Java SE 1.4中，可以有一种更好的处理方法，并且将原始异常设置为新异常的“诱饵”：

```
try
{
   access the database
}
catch (SQLException e)
{
   Throwable se = new ServletException("database error");
   se.initCause(e);
   throw se;
}
```

当捕获到异常时，就可以使用下面这条语句重新得到原始异常：

```
Throwable e = se.getCause();
```

强烈地建议使用这种包装技术。这样可以让用户抛出子系统中的高级异常，而不会丢失原始异常的细节。

提示：如果在一个方法中发生了一个已检查异常，而不允许抛出它，那么包装技术就十分有用。我们可以捕获这个已检查异常，并将它包装成一个运行时异常。

注释：有些异常类，例如，ClassNotFoundException、InvocationTargetException和RuntimeException，拥有它们自己的异常链方案。在Java SE1.4中，这些已经引入“诱饵”机制。然而，仍然可以利用原来的方式，或者调用getCause得到异常链。

11.2.3 Finally子句

当代码抛出一个异常时，就会终止方法中剩余代码的处理，并退出这个方法的执行。如果方法获得了一些本地资源，并且只有这个方法自己知道，又如果这些资源在退出方法之前必须被回收，那么就会产生资源回收问题。一种解决方案是捕获并重新抛出所有的异常。但是，这种解决方案比较乏味，这是因为需要在两个地方清除所分配的资源。一个在正常的代码中，另一个在异常代码中。

Java有一种更好的解决方案，这就是finally子句。下面将介绍如何恰当地释放Graphics对象。如果使用Java编写数据库程序，就需要使用这个技术关闭与数据库的连接。在卷II的第4章中可以看到更加详细的介绍。当发生异常时，恰当地关闭所有数据库的连接是非常重要的。

不管是否有异常被捕获，fianlly子句中的代码都被执行。在下面的示例中，程序将释放所有环境中的图形设备文本。

```
Graphics g = image.getGraphics();
try
{
   // 1
```

```
    code that might throw exceptions
    // 2
}
catch (IOException e)
{
    // 3
    show error dialog
    // 4
}
finally
{
    // 5
    g.dispose();
}
// 6
```

在上面这段代码中，有下列三种情况会执行finally子句：

1）代码没有抛出异常。在这种情况下，程序首先执行try语句块中的全部代码，然后执行finally子句中的代码。随后，继续执行try语句块之后的第一条语句。也就是说，执行标注的1、2、5、6处。

2）抛出一个在catch子句中捕获的异常。在上面的示例中就是IOException异常。在这种情况下，程序将执行try语句块中的所有代码，直到发生异常为止。此时，将跳过try语句块中的剩余代码，转去执行与该异常匹配的catch子句中的代码，最后执行finally子句中的代码。

如果catch子句没有抛出异常，程序将执行try语句块之后的第一条语句。在这里，执行标注1、3、4、5、6处的语句。

如果catch子句抛出了一个异常，异常将被抛回这个方法的调用者。在这里，执行标注1、3、5处的语句。

3）代码抛出了一个异常，但这个异常不是由catch子句捕获的。在这种情况下，程序将执行try语句块中的所有语句，直到有异常被抛出为止。此时，将跳过try语句块中的剩余代码，然后执行finally子句中的语句，并将异常抛给这个方法的调用者。在这里，执行标注1、5处的语句。

try语句可以只有finally子句，而没有catch子句。例如，下面这条try语句：

```
InputStream in = ...;
try
{
    code that might throw exceptions
}
finally
{
    in.close();
}
```

无论在try语句块中是否遇到异常，fianlly子句中的in.close() 语句都会被执行。当然，如果真的遇到一个异常，这个异常将会被重新抛出，并且必须由另一个catch子句捕获。

事实上，我们认为在需要关闭资源时，用这种方式使用fianlly子句是一种不错的选择。下面的提示将给予具体的解释。

提示：这里，强烈建议独立使用try/catch和try/finally语句块。这样可以提高代码的清晰度。例如，

```
InputStream in = ...;
try
{
   try
   {
      code that might throw exceptions
   }
   finally
   {
      in.close();
   }
}
catch (IOException e)
{
   show error dialog
}
```

内层的try语句块只有一个职责，就是确保关闭输入流。外层的try语句块也只有一个职责，就是确保报告出现的错误。这种设计方式不仅清楚，而且还具有一个功能，就是将会报告fianlly子句中出现的错误。

警告：当finally子句包含return语句时，将会出现一种意想不到的结果。假设利用return语句从try语句块中退出。在方法返回前，finally子句的内容将被执行。如果finally子句中也有一个return语句，这个返回值将会覆盖原始的返回值。请看一个例子：

```
public static int f(int n)
{
   try
   {
      int r = n * n;
      return r;
   }
   finally
   {
      if (n == 2) return 0;
   }
}
```

如果调用f(2)，那么try语句块的计算结果为r = 4，并执行return语句。然而，在方法真正返回前，还要执行finally子句。finally子句将使得方法返回0，这个返回值覆盖了原始的返回值4。

有时候，finally子句也会带来麻烦。例如，回收资源的方法也有可能抛出异常。一个比较典型的例子是关闭流（有关流的详细内容将在第12章介绍）。假设希望能够确保在流处理代码中遇到异常时将流关闭。

```
InputStream in = ...;
try
{
   code that might throw exceptions
```

```
}
finally
{
   in.close();
}
```

现在，假设在try语句块中的代码抛出了一些非IOException的异常，这些异常只有这个方法的调用者才能够给予处理。执行finally语句块，并调用close方法。而close方法本身也有可能抛出IOException异常。当出现这种情况时，原始的异常将会丢失，转而抛出IOException异常。这并不是异常处理机制所希望的结果。

解决这个问题的最好方法是在用户调用的finally子句中执行清除操作时不要抛出异常，例如，dispose、close等等，但是InputStream类的设计者并没有这样做。

C++注释：在异常处理方面，C++与Java存在重要区别。Java没有析构器，因此，没有C++的自动回收功能。这就意味着Java程序员必须手工地将回收资源的代码写入finally子句中。当然，由于Java内置了垃圾回收机制，所以，只有极少数的资源需要人工回收。

11.2.4 分析堆栈跟踪元素

堆栈跟踪（stack trace）是一个方法调用过程的列表，它包含了程序执行过程中方法调用的特定位置。前面已经看到过这种列表，当Java程序正常终止，而没有捕获异常时，这个列表就会显示出来。

在Java SE 1.4以前的版本中，可以调用Throwable类的printStackTrace方法访问堆栈跟踪的文本描述信息。现在，可以调用getStackTrace方法获得一个StatckTraceElement对象的数组，并在程序中对它进行分析。例如，

```
Throwable t = new Throwable();
StackTraceElement[] frames = t.getStackTrace();
for (StackTraceElement frame : frames)
   analyze frame
```

StackTraceElement类含有能够获得文件名和当前执行的代码行号的方法，同时，还含有能够获得类名和方法名的方法。toString方法将产生一个格式化的字符串，其中包含所获得的信息。

在Java SE 5.0中，增加了一个静态的Thread.getAllStackTrace方法，它可以产生所有线程的堆栈跟踪。下面给出使用这个方法的具体方式：

```
Map<Thread, StackTraceElement[]> map = Thread.getAllStackTraces();
for (Thread t : map.keySet())
{
   StackTraceElement[] frames = map.get(t);
   analyze frames
}
```

有关Map接口与线程的更加详细的信息请参看第13章和第14章。

例11-1打印了递归阶乘的堆栈情况。例如，如果计算factorial(3)，将会打印下列内容：

```
factorial(3):
StackTraceTest.factorial(StackTraceTest.java:18)
StackTraceTest.main(StackTraceTest.java:34)
factorial(2):
```

```
StackTraceTest.factorial(StackTraceTest.java:18)
StackTraceTest.factorial(StackTraceTest.java:24)
StackTraceTest.main(StackTraceTest.java:34)
factorial(1):
StackTraceTest.factorial(StackTraceTest.java:18)
StackTraceTest.factorial(StackTraceTest.java:24)
StackTraceTest.factorial(StackTraceTest.java:24)
StackTraceTest.main(StackTraceTest.java:34)
return 1
return 2
return 6
```

例11-1 StackTraceTest.java

```java
import java.util.*;

/**
 * A program that displays a trace feature of a recursive method call.
 * @version 1.01 2004-05-10
 * @author Cay Horstmann
 */
public class StackTraceTest
{
   /**
    * Computes the factorial of a number
    * @param n a nonnegative integer
    * @return n! = 1 * 2 * . . . * n
    */
   public static int factorial(int n)
   {
      System.out.println("factorial(" + n + "):");
      Throwable t = new Throwable();
      StackTraceElement[] frames = t.getStackTrace();
      for (StackTraceElement f : frames)
         System.out.println(f);
      int r;
      if (n <= 1) r = 1;
      else r = n * factorial(n - 1);
      System.out.println("return " + r);
      return r;
   }

   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);
      System.out.print("Enter n: ");
      int n = in.nextInt();
      factorial(n);
   }
}
```

API java.lang.Throwable 1.0

- `Throwable(Throwable cause)` 1.4
- `Throwable(String message, Throwable cause)` 1.4

用给定的“诱饵”构造一个Throwable对象。

- `Throwable initCause(Throwable cause) 1.4`

  将这个对象设置为“诱饵”。如果这个对象已经被设置为“诱饵”，则抛出一个异常。返回this引用。

- `Throwable getCause() 1.4`

  获得设置为这个对象的“诱饵”的异常对象。如果没有设置“诱饵”，则返回null。

- `StackTraceElement[] getStackTrace() 1.4`

  获得构造这个对象时调用堆栈的跟踪。

API java.lang.Exception 1.0

- `Exception(Throwable cause) 1.4`
- `Exception(String message, Throwable cause)`

  用给定的“诱饵”构造一个异常对象。

API java.lang.RuntimeException 1.0

- `RuntimeException(Throwable cause) 1.4`
- `RuntimeException(String message, Throwable cause) 1.4`

  用给定的“诱饵”构造一个RuntimeException对象。

API java.lang.StackTraceElement 1.4

- `String getFileName()`

  返回这个元素运行时对应的源文件名。如果这个信息不存在，则返回null。

- `int getLineNumber()`

  返回这个元素运行时对应的源文件行数。如果这个信息不存在，则返回-1。

- `String getClassName()`

  返回这个元素运行时对应的类的全名。

- `String getMethodName()`

  返回这个元素运行时对应的方法名。构造器名是<int>；静态初始化器名是<clinit>。这里无法区分同名的重载方法。

- `boolean isNativeMethod()`

  如果这个元素运行时在一个本地方法中，则返回true。

- `String toString()`

  如果存在的话，返回一个包含类名、方法名、文件名和行数的格式化字符串。

11.3 使用异常机制的建议

目前，存在着大量有关如何恰当地使用异常机制的争论。有些程序员认为所有的已检查异常都很令人厌恶；还有一些程序员认为能够抛出的异常量不够。我们认为异常机制（甚至是已

检查异常）有其用武之地。下面给出使用异常机制的几点建议。

1. 异常处理不能代替简单的测试

作为一个示例，在这里编写了一段应用内置stack类的代码，试着上百万次地对一个空栈进行退栈操作。在实施退栈操作之前，首先要查看栈是否为空。

```
if (!s.empty()) s.pop();
```

接下来，强行进行退栈操作。然后，捕获EmptyStackException异常来告知我们不能这样做。

```
try()
{
   s.pop();
}
catch (EmptyStackException e)
{
}
```

在测试的机器上，得到了表11-1中的时间数据。

表11-1 时间数据

Test	Throw/Catch
646 milliseconds	21,739 milliseconds

可以看出，与执行简单的测试相比，捕获异常所花费的时间大大超过了前者，因此使用异常的基本规则是：只在异常情况下使用异常机制。

2. 不要过分地细化异常

很多程序员习惯将每一条语句都分装在一个独立的try语句块中。

```
OutputStream out;
Stack s;

for (i = 0; i < 100; i++)
{
   try
   {
      n = s.pop();
   }
   catch (EmptyStackException s)
   {
      // stack was empty
   }
   try
   {
      out.writeInt(n);
   }
   catch (IOException e)
   {
      // problem writing to file
   }
}
```

这种编程方式将导致代码量的急剧膨胀。首先看一下这段代码所完成的任务。在这里，希望从栈中弹出100个数值，然后将它们存入一个文件中。如果栈是空的，则不会变成非空状态；如果文件出现错误，则也很难给予排除。出现上述问题后，这种编程方式无能为力。因此，有必要将整个任务包装在一个try语句块中，这样，当任何一个操作出现问题时，整个任务都可

以被取消。

```
try
{
   for (i = 0; i < 100; i++)
   {
      n = s.pop();
      out.writeInt(n);
   }
}
catch (IOException e)
{
   // problem writing to file
}
catch (EmptyStackException s)
{
   // stack was empty
}
```

这段代码看起来清晰多了。这样也满足了异常处理机制的其中一个目标，将正常处理与错误处理分开。

3. 利用异常层次结构

不要只抛出RuntimeException异常。应该寻找更加适当的子类或创建自己的异常类。

不要只捕获Thowable异常，否则，会使程序代码更难读、更难维护。

研究一下已检查异常与未检查异常的区别。已检查异常本来就很庞大，这里不会抛出由逻辑错误造成的异常。例如，反射库发生了错误，而调用者却经常需要捕获那些早已知道不可能发生的异常。

将一种异常转换成另一种更加适合的异常时不要犹豫。例如，在解析某个文件中的一个整数时，捕获NumberFormatException异常，然后将它转换成IOException 或MySubsystemException的子类。

4. 不要压制异常

在Java中，存在着强烈的关闭异常的倾向。如果编写了一个调用一个方法的方法，而这个方法有可能100年才抛出一个异常，那么，编译器会因为没有将这个异常列在throws表中产生抱怨。而没有将这个异常列在throws表中主要出于编译器将会对所有调用这个方法的方法进行异常处理的考虑。因此，应该将这个异常关闭：

```
public Image loadImage(String s)
{
   try
   {
      code that threatens to throw checked exceptions
   }
   catch (Exception e)
   {} // so there
}
```

现在，这段代码就可以通过编译了。除非发生异常，否则它将可以正常地运行。即使发生了异常也会被忽略。如果认为异常非常重要，就应该对它们进行处理。

5. *在检测错误时，"苛刻"要比放任更好*

当检测到错误的时候，有些程序员担心抛出异常。在用无效的参数调用一个方法时，返回一个虚拟的数值，还是抛出一个异常，哪种处理方式更好？例如，当栈空时，Stack.pop是返回一个null，还是抛出一个异常？我们认为:在出错的地方抛出一个EmptyStackException异常要比在后面抛出一个NullPointerException异常更好。

6. *不要羞于传递异常*

很多程序员都感觉应该捕获抛出的全部异常。如果调用了一个抛出异常的方法，例如，FileInputStream 构造器或readLine方法，这些方法就会本能地捕获这些可能产生的异常。其实，传递异常要比捕获这些异常更好：

```
public void readStuff(String filename) throws IOException // not a sign of shame!
{
   InputStream in = new FileInputStream(filename);
   . . .
}
```

让高层次的方法通告用户发生了错误，或者放弃不成功的命令更加适宜。

✔ **注释**：规则5、8可以归纳为"早抛出，晚捕获"。

11.4 断言

在一个具有自我保护能力的程序中，断言是一个常用的习语。假设确信某个属性符合要求，并且代码的执行依赖于这个属性。例如，需要计算

```
double y = Math.sqrt(x);
```

我们确信，这里的x是一个非负数值。原因是：x是另外一个计算的结果，而这个结果不可能是负值；或者x是一个方法的参数，而这个方法要求它的调用者只能提供一个正整数。然而，还是希望进行检查，以避免让错误的数值参与计算操作。当然，也可以抛出一个异常：

```
if (x < 0) throw new IllegalArgumentException("x < 0");
```

但是这段代码会一直保留在程序中，即使测试完毕也不会自动地删除。如果在程序中含有大量的这种检查，程序运行起来会相当慢。

断言机制允许在测试期间向代码中插入一些检查语句。当代码发布时，这些插入的检测语句将会被自动地移走。

在Java SE 1.4中，Java语言引入了关键字assert。这个关键字有两种形式：

```
assert 条件;
```

和

```
assert 条件：表达式;
```

这两种形式都会对条件进行检测，如果结果为false，则抛出一个AssertionError异常。在第二种形式中，表达式将被传入AssertionError的构造器，并转换成一个消息字符串。

注释："表达式"部分的惟一目的是产生一个消息字符串。AssertionErro对象并不存储表达式的值，因此，不可能在以后得到它。正如JDK文档所描述的那样：如果使用表达式的值，就会鼓励程序员试图从断言中恢复程序的运行，这不符合断言机制的初衷。

要想断言x是一个非负数值，只需要简单地使用下面这条语句

```
assert x >= 0;
```

或者将x的实际值传递给AssertionError对象，从而可以在后面显示出来。

```
assert x >= 0 : x;
```

C++注释：C语言中的assert 宏将断言中的条件转换成一个字符串。当断言失败时，这个字符串将会被打印出来。例如，若assert(x>=0)失败，那么将打印出失败条件"x>=0"。在Java中，条件并不会自动地成为错误报告中的一部分。如果希望看到这个条件，就必须将它以字符串的形式传递给AssertionError对象：assert x >= 0 : "x >= 0"。

11.4.1 启用和禁用断言

在默认情况下，断言被禁用。可以在运行程序时用-enableassertions或-ea选项启用它：

```
java -enableassertions MyApp
```

需要注意：在启用或禁用断言时不必重新编译程序。启用或禁用断言是*类加载器*（class loader）的功能。当断言被禁用时，类加载器将跳过断言代码，因此，不会降低程序运行的速度。

也可以在某个类或某个包中使用断言。例如，

```
java -ea:MyClass -ea:com.mycompany.mylib... MyApp
```

这条命令将开启MyClass类以及在com.mycompany.mylib包和它的子包中的所有类的断言。选项-ea将开启默认包中的所有类的断言。

也可以用选项-disableassertions或-da禁用某个特定类和包的断言：

```
java -ea:... -da:MyClass MyApp
```

有些类不是由类加载器加载，而是直接由虚拟机加载。可以使用这些开关有选择地启用或禁用那些类中的断言。然而，启用和禁用所有断言的-ea和-da开关不能应用到那些没有类加载器的"系统类"上。对于这些系统类来说，需要使用-enablesystemassertions/-esa开关启用断言。

在程序中也可以控制类加载器的断言状态。有关这方面的内容请参看本节末尾的API注解。

11.4.2 使用断言的建议

在Java语言中，给出了三种处理系统错误的机制：

- 抛出一个异常
- 日志
- 使用断言

什么时候应该选择使用断言呢？请记住下面几点：

- 断言失败是致命的、不可恢复的错误。
- 断言检查只用于开发和测阶段（这种做法有时候被戏称为"在靠近海岸时穿上救生衣，但在海中央时就把救生衣抛入水中吧"）。

因此，不应该使用断言向程序的其他部分通告发生了可恢复性的错误，或者，不应该作为程序向用户通告问题的手段。断言只应该用于在测试阶段确定程序内部的错误位置。

下面看一个十分常见的例子：检查方法的参数。是否应该使用断言来检查非法的下标值或null引用呢？要想回答这个问题，首先阅读一下这个方法的文档。假设实现一个排序方法。

```
/**
   Sorts the specified range of the specified array into ascending numerical order.
   The range to be sorted extends from fromIndex, inclusive, to toIndex, exclusive.
   @param a the array to be sorted.
   @param fromIndex the index of the first element (inclusive) to be sorted.
   @param toIndex the index of the last element (exclusive) to be sorted.
   @throws IllegalArgumentException if fromIndex > toIndex
   @throws ArrayIndexOutOfBoundsException if fromIndex < 0 or toIndex > a.length
*/
static void sort(int[] a, int fromIndex, int toIndex)
```

文档说明，如果方法中使用了错误的下标值，那么就会抛出一个异常。这是方法与调用者之间约定的处理行为。如果实现这个方法，那就必须要遵守这个约定，并抛出表示下标值有误的异常。因此，这里使用断言不太适宜。

是否应该断言a而不是null吗？这也不太适宜。当a是null时，这个方法的文档没有指出应该采取什么行动。在这种情况下，调用者可以认为这个方法将会成功地返回，而不会抛出一个断言错误。

然而，假设对这个方法的约定做一点微小的改动：

```
@param a the array to be sorted. (Must not be null)
```

现在，这个方法的调用者就必须注意：不允许用null数组调用这个方法，并在这个方法的开头使用断言

```
assert a != null;
```

计算机科学家将这种约定称为前提条件（Precondition）。最初的方法对参数没有前提条件，即承诺在任何条件下都能够给予正确的执行。修订后的方法有一个前提条件，即a非空。如果调用者在调用这个方法时没有提供满足这个前提条件的参数，所有的断言都会失败，并且这个方法可以执行它想做的任何操作。事实上，由于可以使用断言，当方法被非法调用时，将会出现难以预料的结果。有时候会抛出一个断言错误，有时候会产生一个null指针异常，这完全取决于类加载器的配置。

11.4.3 为文档使用断言

很多程序员使用注释说明假设条件。看一下http://java.sun.com/javase/6/docs/technotes/guides/language/assert.html上的一个示例：

```
if (i % 3 == 0)
   . . .
else if (i % 3 == 1)
   . . .
else // (i % 3 == 2)
   . . .
```

在这个示例中，使用断言会更好一些。

```
if (i % 3 == 0)
   . . .
else if (i % 3 == 1)
   . . .
else
{
   assert i % 3 == 2;
   . . .
}
```

当然，如果再仔细地考虑一下这个问题会发现一个更有意思的内容。i%3会产生什么结果？如果i是正值，那余数肯定是0、1或2。如果i是负值，则余数则可以是－1和－2。然而，实际上都认为i是非负值，因此，最好在if语句之前使用下列断言：

```
assert i >= 0;
```

无论如何，这个示例说明了程序员如何使用断言来进行自我检查。前面已经知道，断言是一种测试和调试阶段所使用的战术性工具；而日志记录是一种在程序的整个生命周期都可以使用的策略性工具。稍后将介绍日志的相关知识。

API java.lang.ClassLoader 1.0

- `void setDefaultAssertionStatus(boolean b) 1.4`

  对于通过类加载器加载的所有类来说，如果没有显式地说明类或包的断言状态，就启用或禁用断言。
- `void setClassAssertionStatus(String className, boolean b) 1.4`

  对于给定的类和它的内部类，启用或禁用断言。
- `void setPackageAssertionStatus(String packageName, boolean b) 1.4`

  对于给定包和其子包中的所有类，启用或禁用断言。
- `void clearAssertionStatus() 1.4`

  移去所有类和包的显式断言状态设置，并禁用所有通过这个类加载器加载的类的断言。

11.5 记录日志

每个Java程序员都很熟悉在有问题的代码中插入一些调用System.out.println方法的语句来帮助观察程序运行的操作过程。当然，一旦发现问题的根源，就要将这些语句从代码中删去。如果接下来又出现了问题，就需要再插入几个调用System.out.println方法的语句。记录日志API就是为了解决这个问题而设计的。下面先讨论一下这些API的优点。

- 可以很容易地取消全部日志记录，或者仅仅取消某个级别的日志，而且打开和关闭这个操作也很容易。
- 可以很简单地禁止日志记录的输出，因此，将这些日志代码留在程序中所付出的代价很小。
- 日志记录可以被定向到不同的处理器，用于在控制台中显示，用于存储在文件中等等。
- 日志记录器和处理器都可以对记录进行过滤。过滤器可以根据过滤实现器制定的标准丢

弃那些无用的记录项。

- 日志记录可以采用不同的方式格式化，例如，纯文本或XML。
- 应用程序可以使用多个日志记录器，它们使用类似包名的这种具有层次结构的名字，例如，com.mycompany.myapp。
- 在默认情况下，日志系统的配置由配置文件控制。如果需要的话，应用程序可以替换这个配置。

11.5.1 基本日志

下面从一个最简单的例子开始。日志系统管理着一个名为Logger.global的默认日志记录器，可以用System.out替换它，并通过调用info方法记录日志信息：

```
Logger.global.info("File->Open menu item selected");
```

在默认情况下，这条记录将会显示出如下所示的内容：

```
May 10, 2004 10:12:15 PM LoggingImageViewer fileOpen
INFO: File->Open menu item selected
```

注意：自动包含了时间、调用的类名和方法名。但是，如果在恰当的地方（例如，main开始）调用

```
Logger.global.setLevel(Level.OFF);
```

将会取消所有的日志。

11.5.2 高级日志

从前面已经看到"虚拟日志"，下面继续看一下企业级（industrial-strength）日志。在一个专业的应用程序中，不要将所有的日志都记录到一个全局日志记录器中，而是可以自定义日志记录器。

当第一次请求一个具有给定名字的日志记录器时，就会创建这个记录器。

```
Logger myLogger = Logger.getLogger("com.mycompany.myapp");
```

如果后面使用同一个名字调用日志记录器就会产生相同的日志记录器对象。

与包名类似，日志记录器名也具有层次结构。事实上，与包名相比，日志记录器的层次性更强。对于包来说，一个包的名字与其父包的名字之间没有语义关系，但是日志记录器的父与子之间将共享某些属性。例如，如果对com.mycompany日志记录器设置了日志级别，它的子记录器也会继承这个级别。

通常，有以下7个日志记录器级别：

- SEVERE
- WARNING
- INFO
- CONFIG
- FINE
- FINER

• FINEST

在默认情况下，只记录前三个级别。也可以设置其他的级别。例如，

```
logger.setLevel(Level.FINE);
```

现在，FINE和更高级别的记录都可以记录下来。

另外，还可以使用Level.ALL开启所有级别的记录，或者使用Level.OFF关闭所有级别的记录。

对于所有的级别有下面几种记录方法：

```
logger.warning(message);
logger.fine(message);
```

同时，还可以使用log方法指定级别，例如，

```
logger.log(Level.FINE, message);
```

提示：默认的日志配置记录了INFO或更高级别的所有记录，因此，应该使用CONFIG、FINE、FINER和FINEST级别来记录那些有助于诊断，但对于程序员又没有太大意义的调试信息。

警告：如果将记录级别设计为INFO或者更低，则需要修改日志处理器的配置。默认的日志处理器不会处理低于INFO级别的信息。有关更加详细的内容请参看下一节。

默认的日志记录将显示包含日志调用的类名和方法名，如同堆栈所显示的那样。但是，如果虚拟机对执行过程进行了优化，就得不到准确的调用信息。此时，可以调用logp方法获得调用类和方法的确切位置，这个方法的签名为：

```
void logp(Level l, String className, String methodName, String message)
```

下面有一些用来跟踪执行流程的方法：

```
void entering(String className, String methodName)
void entering(String className, String methodName, Object param)
void entering(String className, String methodName, Object[] params)
void exiting(String className, String methodName)
void exiting(String className, String methodName, Object result)
```

例如：

```
int read(String file, String pattern)
{
   logger.entering("com.mycompany.mylib.Reader", "read",
      new Object[] { file, pattern });
   . . .
   logger.exiting("com.mycompany.mylib.Reader", "read", count);
   return count;
}
```

这些调用将生成FINER级别和以字符串“ENTRY”和“RETURN”开始的日志记录。

注释：在未来，带Object[]参数的日志记录方法可能会被重写，以便支持变量参数列表（“varargs”）。此后就可以用logger.entering（“com.mycompany.mylib.Reader”，“read”，file, pattern)格式调用这个方法了。

记录日志的常见用途是记录那些不可预料的异常。可以使用下面两个方法提供日志记录中

包含的异常描述内容。

```
void throwing(String className, String methodName, Throwable t)
void log(Level l, String message, Throwable t)
```

典型的用法是：

```
if (. . .)
{
   IOException exception = new IOException(". . .");
   logger.throwing("com.mycompany.mylib.Reader", "read", exception);
   throw exception;
}
```

还有

```
try
{
   . . .
}
catch (IOException e)
{
   Logger.getLogger("com.mycompany.myapp").log(Level.WARNING, "Reading image", e);
}
```

调用throwing可以记录一条FINER级别的记录和一条以THROW开始的信息。

11.5.3 修改日志管理器配置

可以通过编辑配置文件来修改日志系统的各种属性。在默认情况下，配置文件存在于：

jre/lib/logging.properties

要想使用另一个配置文件，就要将java.util.logging.config.file特性设置为配置文件的存储位置，并用下列命令启动应用程序：

java -Djava.util.logging.config.file=*configFile MainClass*

警告：*在main方法中调用 System.setProperty("java.util.logging.config.file", file)没有任何影响，其原因是日志管理器在VM启动过程中被初始化，它会在main之前执行。*

要想修改默认的日志记录级别，就需要编辑配置文件，并修改以下命令行

```
.level=INFO
```

可以通过添加以下内容来指定自己的日志记录级别

```
com.mycompany.myapp.level=FINE
```

也就是说，在日志记录名后面添加后缀.level。

在稍后可以看到，日志记录并不将消息发送到控制台上，这是处理器的任务。另外，处理器也有级别。要想在控制台上看到FINE级别的消息，就需要进行下列设置

```
java.util.logging.ConsoleHandler.level=FINE
```

警告：*在日志管理器配置的属性设置不是系统属性，因此，用-Dcom.mycompany.myapp.level= FINE启动应用程序不会对日志记录产生任何影响。*

警告：截止到Java SE 6，Logmanager类的API文档主张通过Preferences API设置java.util.logging.config.class和java.util.logging.config.file属性。这是不正确的，有关信息请参看http://bugs.sun.com/bugdatabase/view_bug.di?bug_id=4691587。

注释：日志属性文件由java.util.logging.LogManager类处理。可以通过将java.util.logging.manager系统属性设置为某个子类的名字来指定一个不同的日志管理器。另外，在保存标准日志管理器的同时，还可以从日志属性文件跳过初始化。还有一种方式是将java.util.logging.config.class系统属性设置为某个类名，该类再通过其他方式设定日志管理器属性。有关LogManager类的详细内容请参看API文档。

在运行的程序中，使用jconsole程序也可以改变日志记录的级别。有关信息请参看http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html#LoggingControl。

11.5.4 本地化

我们可能希望将日志消息本地化，以便让全球的用户都可以阅读它。应用程序的国际化问题将在卷II的第5章中讨论。下面简要地说明一下在本地化日志消息时需要牢记的一些要点。

本地化的应用程序包含资源包（resource bundle）中的本地说明信息。资源包由各个地区（例如，美国、德国）的映射集合组成。例如，某个资源包可能将字符串“readingFile”映射成英文的“Reading file”或者德文的“Achtung! Datei wird eingelesen”。

一个程序可以包含多个资源包，一个用于菜单；其他用于日志消息。每个资源包都有一个名字（例如，com.mycompany.logmessages）。要想将映射添加到一个资源包中，需要为每个地区创建一个文件。英文消息映射位于com/mycompany/logmessages_en.properties文件中；德文消息映射位于com/mycompany/logmessages_de.properties文件中。（en、de是语言编码）。可以将这些文件与应用程序的类文件放在一起，以便ResourceBundle类自动地对它们进行定位。这些文件都是纯文本文件，其组成项如下所示：

```
readingFile=Achtung! Datei wird eingelesen
renamingFile=Datei wird umbenannt
...
```

在请求日志记录器时，可以指定一个资源包：

```
Logger logger = Logger.getLogger(loggerName, "com.mycompany.logmessages");
```

然后，为日志消息指定资源包的关键字，而不是实际的日志消息字符串。

```
logger.info("readingFile");
```

通常需要在本地化的消息中增加一些参数，因此，消息应该包括占位符{0}、{1}等等。例如，要想在日志消息中包含文件名，就应该用下列方式包括占位符：

```
Reading file {0}.
Achtung! Datei {0} wird eingelesen.
```

然后，通过调用下面的一个方法向占位符传递具体的值：

```
logger.log(Level.INFO, "readingFile", fileName);
logger.log(Level.INFO, "renamingFile", new Object[] { oldName, newName });
```

11.5.5 处理器

在默认情况下，日志记录器将记录发送到ConsoleHandler中，并由它输出到System.err 流中。特别是，日志记录器还会将记录发送到父处理器中，而最终的处理器（命名为“”）有一个ConsoleHandler。

与日志记录器一样，处理器也有日志记录级别。对于一个要被记录的日志记录，它的日志记录级别必须高于日志记录器和处理器的阈值。日志管理器配置文件设置的默认控制台处理器的日志记录级别为

```
java.util.logging.ConsoleHandler.level=INFO
```

要想记录FINE级别的日志，就必须修改配置文件中的默认日志记录级别和处理器级别。另外，还可以绕过配置文件，安装自己的处理器。

```
Logger logger = Logger.getLogger("com.mycompany.myapp");
logger.setLevel(Level.FINE);
logger.setUseParentHandlers(false);
Handler handler = new ConsoleHandler();
handler.setLevel(Level.FINE);
logger.addHandler(handler);
```

在默认情况下，日志记录器将记录发送到自己的处理器和父处理器。我们的日志记录器是原始日志记录器（命名为“”）的子类，而原始日志记录器将会把所有等于或高于INFO级别的记录发送到控制台。然而，我们并不想两次看到这些记录。鉴于这个原因，应该将useParentHandlers属性设置为false。

要想将日志记录发送到其他地方，就要添加其他的处理器。日志API为此提供了两个很有用的处理器，一个是FileHandler；另一个是SocketHandler。SocketHandler将记录发送到特定的主机和端口。而更令人感兴趣的是FileHandler，它可以收集文件中的记录。

可以像下面这样直接将记录发送到默认文件的处理器：

```
FileHandler handler = new FileHandler();
logger.addHandler(handler);
```

这些记录被发送到用户主目录的javan.log文件中，n是文件名的惟一编号。如果用户系统没有主目录（例如，在Windows95/98/Me），文件就存储在C:\Window这样的默认位置上。在默认情况下，记录被格式化为XML。下面是一个典型的日志记录的形式：

```
<record>
  <date>2002-02-04T07:45:15</date>
  <millis>1012837515710</millis>
  <sequence>1</sequence>
  <logger>com.mycompany.myapp</logger>
  <level>INFO</level>
  <class>com.mycompany.mylib.Reader</class>
  <method>read</method>
  <thread>10</thread>
  <message>Reading file corejava.gif</message>
</record>
```

可以通过设置日志管理器配置文件中的不同参数（请参看表11-2），或者利用其他的构造

器（请参看本节后面给出的API注释）来修改文件处理器的默认行为。

也有可能不想使用默认的日志记录文件名，因此，应该使用另一种模式，例如，%h/myapp.log（有关模式变量的解释请参看表11-3）。

如果多个应用程序（或者同一个应用程序的多个拷贝）使用同一个日志文件，就应该开启append标志。另外，应该在文件名模式中使用%u，以便每个应用程序创建日志的惟一拷贝。

开启文件循环功能也是一个不错的主意。日志文件以myapp.log.0，myapp.log.1，myapp.log.2，这种循环序列的形式出现。只要文件超出了大小限制，最旧的文件就会被删除，其他的文件将重新命名，同时创建一个新文件，其编号为0。

表11-2 文件处理器配置参数

配置属性	描述	默认
java.util.logging.FileHandler.level	处理器级别	Level.ALL
java.util.logging.FileHandler.append	控制处理器应该追加到一个已经存在的文件尾部；还是应该为每个运行的程序打开一个新文件	false
java.util.logging.FileHandler.limit	在打开另一个文件之前允许写入一个文件的近似最大字节数（0表示无限制）	在FileHandler类中为0（表示无限制）；在默认的日志管理器配置文件中为50000。
java.util.logging.FileHandler.pattern	日志文件名的模式。参看表11-3中的模式变量	%h/java%u.log
java.util.logging.FileHandler.count	在循环序列中的日志记录数量	1（不循环）
java.util.logging.FileHandler.filter	使用的过滤器类	没有使用过滤器
java.util.logging.FileHandler.encoding	使用的字符编码	平台的编码
java.util.logging.FileHandler.formatter	记录格式器	java.util.logging.XMLFormatter

提示： 很多程序员将日志记录作为辅助文档提供给技术支持员工。如果程序的行为有误，用户就可以返回查看日志文件以找到错误的原因。在这种情况下，应该开启“append”标志，或使用循环日志，也可以两个功能同时使用。

表11-3 日志记录文件模式变量

变量	描述
%h	系统属性user.home的值
%t	系统临时目录
%u	用于解决冲突的惟一性编号
%g	为循环日志记录生成的数值。(当使用循环功能且模式不包括%g时，使用后缀%g)
%%	%字符

还可以通过扩展Handler类或 StreamHandler 类自定义处理器。在本节结尾的示例程序中就定义了这样一个处理器。这个处理器将在窗口中显示日志记录（如图11-2所示）。

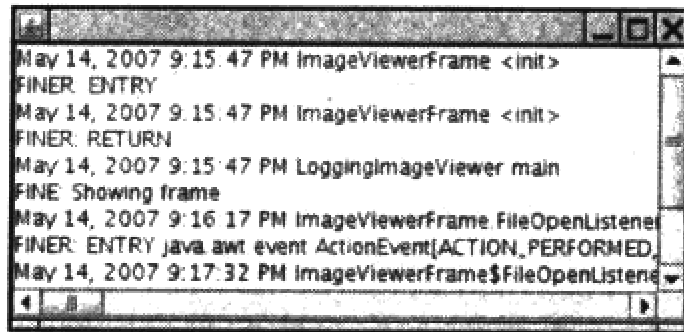


图11-2 在窗口中显示记录的日志处理器

这个处理器扩展于StreamHandler类，并安装了一个流。这个流的write方法将流显示输出到文本框中。

```
class WindowHandler extends StreamHandler
{
   public WindowHandler()
   {
      . . .
      final JTextArea output = new JTextArea();
      setOutputStream(new
         OutputStream()
         {
            public void write(int b) {} // not called
            public void write(byte[] b, int off, int len)
            {
               output.append(new String(b, off, len));
            }
         });
   }
   . . .
}
```

使用这种方式只存在一个问题，这就是处理器会缓存记录，并且只有在缓存满的时候才将它们写入流中，因此，需要覆盖public方法，以便在处理器获得每个记录之后刷新缓冲区。

```
class WindowHandler extends StreamHandler
{
   . . .
   public void publish(LogRecord record)
   {
      super.publish(record);
      flush();
   }
}
```

如果希望编写更加复杂的流处理器，就应该扩展Handler类，并自定义public、flush和close方法。

11.5.6 过滤器

在默认情况下，过滤器根据日志记录的级别进行过滤。每个日志记录器和处理器都可以有一个可选的过滤器来完成附加的过滤。另外，可以通过实现Filter接口并定义下列方法来自定义过滤器。

```
boolean isLoggable(LogRecord record)
```

在这个方法中，可以利用自己喜欢的标准，对日志记录进行分析，返回true表示这些记录应该包含在日志中。例如，某个过滤器可能只对entering方法和exiting方法产生的消息感兴趣，这个过滤器可以调用 record.getMessage() 方法，并查看这个消息是否用ENTRY或RETURN开头。

要想将一个过滤器安装到一个日志记录器或处理器中，只需要调用setFilter方法就可以了。注意，同一时刻最多只能有一个过滤器。

11.5.7 格式化器

ConsoleHandler类和FileHandler类可以生成文本和XML格式的日志记录。但是，也可以自定义格式。这需要扩展Formatter类并覆盖下面这个方法：

```
String format(LogRecord record)
```

可以根据自己的愿望对记录中的信息进行格式化，并返回结果字符串。在format方法中，有可能会调用下面这个方法

```
String formatMessage(LogRecord record)
```

这个方法对记录中的部分消息进行格式化、参数替换和本地化应用操作。

很多文件格式（例如XML）需要在已格式化的记录的前后加上一个头部和尾部。在这个例子中，要覆盖下面两个方法：

```
String getHead(Handler h)
String getTail(Handler h)
```

最后，调用setFormatter方法将格式化器安装到处理器中。

11.5.8 日志记录说明

面对日志记录如此之多的可选项，很容易让人忘记最基本的东西。下面的“日志说明书”总结了一些最常用的操作。

1）为一个简单的应用程序，选择一个日志记录器，并把日志记录器命名为与主应用程序包一样的名字，例如，com.mycompany.myprog，这是一种好的编程习惯。另外，可以通过调用下列方法得到日志记录器。

```
Logger logger = Logger.getLogger("com.mycompany.myprog");
```

为了方便起见，可能希望利用一些日志操作将下面的静态域添加到类中：

```
private static final Logger logger = Logger.getLogger("com.mycompany.myprog");
```

2）默认的日志配置将级别等于或高于INFO级别的所有消息记录到控制台。用户可以覆盖默认的配置文件。但是正如前面所述，改变配置需要做相当多的工作。因此，最好在应用程序中安装一个更加适宜的默认配置。

下列代码确保将所有的消息记录到应用程序特定的文件中。可以将这段代码放置在应用程序的main方法中。

```
if (System.getProperty("java.util.logging.config.class") == null
      && System.getProperty("java.util.logging.config.file") == null)
```

```
{
   try
   {
      Logger.getLogger("").setLevel(Level.ALL);
      final int LOG_ROTATION_COUNT = 10;
      Handler handler = new FileHandler("%h/myapp.log", 0,  LOG_ROTATION_COUNT);
      Logger.getLogger("").addHandler(handler);
   }
   catch (IOException e)
   {
      logger.log(Level.SEVERE, "Can't create log file handler", e);
   }
}
```

3）现在，可以记录自己想要的内容了。但需要牢记：所有级别为INFO、WARNING和SEVERE的消息都将显示到控制台上。因此，最好只将对程序用户有意义的消息设置为这几个级别。将程序员想要的日志记录，设定为FINE是一个很好的选择。

当调用System.out.println时，实际上生成了下面的日志消息：

```
logger.fine("File open dialog canceled");
```

记录那些不可预料的异常也是一个不错的想法。例如，

```
try
{
   . . .
}
catch (SomeException e)
{
   logger.log(Level.FINE, "explanation", e);
}
```

例11-2利用上述说明可实现：日志记录消息也显示在日志窗口中。

例11-2 LoggingImageViewer.java

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.logging.*;
import javax.swing.*;

/**
 * A modification of the image viewer program that logs various events.
 * @version 1.02 2007-05-31
 * @author Cay Horstmann
 */
public class LoggingImageViewer
{
   public static void main(String[] args)
   {
      if (System.getProperty("java.util.logging.config.class") == null
            && System.getProperty("java.util.logging.config.file") == null)
      {
         try
         {
            Logger.getLogger("com.horstmann.corejava").setLevel(Level.ALL);
```

```
            final int LOG_ROTATION_COUNT = 10;
            Handler handler = new FileHandler("%h/LoggingImageViewer.log", 0, LOG_ROTATION_COUNT);
            Logger.getLogger("com.horstmann.corejava").addHandler(handler);
         }
         catch (IOException e)
         {
            Logger.getLogger("com.horstmann.corejava").log(Level.SEVERE,
                  "Can't create log file handler", e);
         }
      }

      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               Handler windowHandler = new WindowHandler();
               windowHandler.setLevel(Level.ALL);
               Logger.getLogger("com.horstmann.corejava").addHandler(windowHandler);

               JFrame frame = new ImageViewerFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

               Logger.getLogger("com.horstmann.corejava").fine("Showing frame");
               frame.setVisible(true);
            }
         });
   }
}

/**
 * The frame that shows the image.
 */
class ImageViewerFrame extends JFrame
{
   public ImageViewerFrame()
   {
      logger.entering("ImageViewerFrame", "<init>");
      setTitle("LoggingImageViewer");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // set up menu bar
      JMenuBar menuBar = new JMenuBar();
      setJMenuBar(menuBar);

      JMenu menu = new JMenu("File");
      menuBar.add(menu);

      JMenuItem openItem = new JMenuItem("Open");
      menu.add(openItem);
      openItem.addActionListener(new FileOpenListener());

      JMenuItem exitItem = new JMenuItem("Exit");
      menu.add(exitItem);
      exitItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
```

```
               logger.fine("Exiting.");
               System.exit(0);
            }
         });

      // use a label to display the images
      label = new JLabel();
      add(label);
      logger.exiting("ImageViewerFrame", "<init>");
   }

   private class FileOpenListener implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         logger.entering("ImageViewerFrame.FileOpenListener", "actionPerformed", event);

         // set up file chooser
         JFileChooser chooser = new JFileChooser();
         chooser.setCurrentDirectory(new File("."));

         // accept all files ending with .gif
         chooser.setFileFilter(new javax.swing.filechooser.FileFilter()
            {
               public boolean accept(File f)
               {
                  return f.getName().toLowerCase().endsWith(".gif") || f.isDirectory();
               }

               public String getDescription()
               {
                  return "GIF Images";
               }
            });

         // show file chooser dialog
         int r = chooser.showOpenDialog(ImageViewerFrame.this);

         // if image file accepted, set it as icon of the label
         if (r == JFileChooser.APPROVE_OPTION)
         {
            String name = chooser.getSelectedFile().getPath();
            logger.log(Level.FINE, "Reading file {0}", name);
            label.setIcon(new ImageIcon(name));
         }
         else logger.fine("File open dialog canceled.");
         logger.exiting("ImageViewerFrame.FileOpenListener", "actionPerformed");
      }
   }

   private JLabel label;
   private static Logger logger = Logger.getLogger("com.horstmann.corejava");
   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 400;
}

/**
```

```
 * A handler for displaying log records in a window.
 */
class WindowHandler extends StreamHandler
{
   public WindowHandler()
   {
      frame = new JFrame();
      final JTextArea output = new JTextArea();
      output.setEditable(false);
      frame.setSize(200, 200);
      frame.add(new JScrollPane(output));
      frame.setFocusableWindowState(false);
      frame.setVisible(true);
      setOutputStream(new OutputStream()
         {
            public void write(int b)
            {
            } // not called

            public void write(byte[] b, int off, int len)
            {
               output.append(new String(b, off, len));
            }
         });
   }

   public void publish(LogRecord record)
   {
      if (!frame.isVisible()) return;
      super.publish(record);
      flush();
   }

   private JFrame frame;
}
```

API java.util.logging.Logger 1.4

- `Logger getLogger(String loggerName)`
- `Logger getLogger(String loggerName, String bundleName)`

  获得给定名字的日志记录器。如果这个日志记录器不存在，创建一个日志记录器。

  参数：loggerName　具有层次结构的日志记录器名。例如，com.mycompany.myapp

  　　　bundleName　用来查看本地消息的资源包名。

- `void severe(String message)`
- `void warning(String message)`
- `void info(String message)`
- `void config(String message)`
- `void fine(String message)`
- `void finer(String message)`
- `void finest(String message)`

记录一个由方法名和给定消息指示级别的日志记录。

- void entering(String className, String methodName)
- void entering(String className, String methodName, Object param)
- void entering(String className, String methodName, Object[] param)
- void exiting(String className, String methodName)
- void exiting(String className, String methodName, Object result)

记录一个描述进入/退出方法的日志记录，其中应该包括给定参数和返回值。

- void throwing(String className, String methodName, Throwable t)

记录一个描述抛出给定异常对象的日志记录。

- void log(Level level, String message)
- void log(Level level, String message, Object obj)
- void log(Level level, String message, Object[] objs)
- void log(Level level, String message, Throwable t)

记录一个给定级别和消息的日志记录，其中可以包括对象或者可抛出对象。要想包括对象，消息中必须包含格式化占位符{0}、{1}等。

- void logp(Level level, String className, String methodName, String message)
- void logp(Level level, String className, String methodName, String message, Object obj)
- void logp(Level level, String className, String methodName, String message, Object[] objs)
- void logp(Level level, String className, String methodName, String message, Throwable t)

记录一个给定级别、准确的调用者信息和消息的日志记录，其中可以包括对象或可抛出对象。

- void logrb(Level level, String className, String methodName, String bundleName, String message)
- void logrb(Level level, String className, String methodName, String bundleName, String message, Object obj)
- void logrb(Level level, String className, String methodName, String bundleName, String message, Object[] objs)
- void logrb(Level level, String className, String methodName, String bundleName, String message, Throwable t)

记录一个给定级别、准确的调用者信息、资源包名和消息的日志记录，其中可以包括对象或可抛出对象。

- Level getLevel()
- void setLevel(Level l)

获得和设置这个日志记录器的级别。

- Logger getParent()
- void setParent(Logger l)

  获得和设置这个日志记录器的父日志记录器。
- Handler[] getHandlers()

  获得这个日志记录器的所有处理器。
- void addHandler(Handler h)
- void removeHandler(Handler h)

  增加或删除这个日志记录器中的一个处理器。
- boolean getUseParentHandlers()
- void setUseParentHandlers(boolean b)

  获得和设置"use parent handler"属性。如果这个属性是true，则日志记录器会将全部的日志记录转发给它的父处理器。
- Filter getFilter()
- void setFilter(Filter f)

  获得和设置这个日志记录器的过滤器。

API java.util.logging.Handler 1.4

- abstract void publish(LogRecord record)

  将日志记录发送到希望的目的地。
- abstract void flush()

  刷新所有已缓冲的数据。
- abstract void close()

  刷新所有已缓冲的数据，并释放所有相关的资源。
- Filter getFilter()
- void setFilter(Filter f)

  获得和设置这个处理器的过滤器。
- Formatter getFormatter()
- void setFormatter(Formatter f)

  获得和设置这个处理器的格式化器。
- Level getLevel()
- void setLevel(Level l)

  获得和设置这个处理器的级别。

API java.util.logging.ConsoleHandler 1.4

- ConsoleHandler()

  构造一个新的控制台处理器。

API java.util.logging.FileHandler 1.4

- `FileHandler(String pattern)`
- `FileHandler(String pattern, boolean append)`
- `FileHandler(String pattern, int limit, int count)`
- `FileHandler(String pattern, int limit, int count, boolean append)`

  构造一个文件处理器。

  参数：pattern　构造日志文件名的模式。参见表11-3列出的模式变量
  　　　limit　在打开一个新日志文件之前，日志文件可以包含的近似最大字节数
  　　　count　循环序列的文件数量
  　　　append　新构造的文件处理器对象应该追加在一个已存在的日志文件尾部，则为true

API java.util.logging.LogRecord 1.4

- `Level getLevel()`

  获得这个日志记录的记录级别。
- `String getLoggerName()`

  获得正在记录这个日志记录的日志记录器的名字。
- `ResourceBundle getresourceBundle()`
- `String getresourceBundleName()`

  获得用于本地化消息的资源包或资源包的名字。如果没有获得，则返回null。
- `String getMessage()`

  获得本地化和格式化之前的原始消息。
- `Object[] getParameters()`

  获得参数对象。如果没有获得，则返回null。
- `Throwable getThrown()`

  获得被抛出的对象。如果不存在，则返回null。
- `String getSourceClassName()`
- `String getSourceMethodName()`

  获得记录这个日志记录的代码区域。这个信息有可能是由日志记录代码提供的，也有可能是自动从运行时堆栈推测出来的。如果日志记录代码提供的值有误，或者运行时代码由于被优化而无法推测出确切的位置，这两个方法的返回值就有可能不准确。
- `long getMillis()`

  获得创建时间。以毫秒为单位（从1970年开始）。
- `long getSequenceNumber()`

  获得这个日志记录的惟一序列序号。
- `int getThreadID()`

获得创建这个日志记录的线程的惟一ID。这些ID是由LogRecord类分配的，并且与其他线程的ID无关。

API java.util.logging.Filter 1.4

- `boolean isLoggable(LogRecord record)`

  如果给定日志记录需要记录，则返回true。

API java.util.logging.Formatter 1.4

- `abstract String format(LogRecord record)`

  返回对日志记录格式化后得到的字符串。
- `String getHead(Handler h)`
- `String getTail(Handler h)`

  返回应该出现在包含日志记录的文档的开头和结尾的字符串。超类Formatter定义了这些方法，它们只返回空字符串。如果必要的话，可以对它们进行重载。
- `String formatMessage(LogRecord record)`

  返回经过本地化和格式化后的日志记录的消息内容。

11.6 调试技术

假设编写了一个程序，并对所有的异常进行了捕获和恰当的处理，然后，运行这个程序，但还是出现问题，现在该怎么办呢（如果从来没有遇到过这种情况，可以跳过本章的剩余部分）？

当然，如果有一个方便且功能强大的调试器就太好了。调试器是Eclipse、NetBeans这类专业集成开发环境的一部分。本章稍后将讨论调试器。本节，在启动调试器之前，先给出一些有价值的建议。

1）可以用下面的方法打印或记录任意变量的值：

```
System.out.println("x=" + x);
```

或者

```
Logger.global.info("x=" + x);
```

如果x是一个数值，则会被转换成等价的字符串。如果x是一个对象，那么Java就会调用这个对象的toString方法。要想获得隐式参数对象的状态，就可以打印这个对象的状态。

```
Logger.global.info("this=" + this);
```

Java类库中的绝大多数类都覆盖了toString方法，以便能够提供有用的类信息。这样会使调试更加便捷。在自定义的类中，也应该这样做。

2）一个不太为人所知，但却非常有效的技巧是在每一个类中放置一个main方法。这样就可以对每一个类进行单元测试。

```
public class MyClass
{
   methods and fields
```

```
    . . .
    public static void main(String[] args)
    {
       test code
    }
}
```

利用这种技巧，只需要创建少量的对象，调用所有的方法，并检测每个方法是否能够正确地运行就可以了。另外，可以为每个类保留一个main方法，然后分别为每个文件调用Java虚拟机进行运行测试。在运行applet应用程序的时候，这些main方法不会被调用，而在运行应用程序的时候，Java虚拟机只调用启动类的main方法。

3）如果喜欢使用前面所讲述的技巧，就应该到http://junit.org网站上查看一下JUnit。JUnit是一个非常常见的单元测试框架，利用它可以很容易地组织几套测试用例。只要修改类，就需要运行测试。在发现bug时，还要补充一些其他的测试用例。

4）日志代理（logging proxy）是一个子类的对象，它可以窃取方法调用，并进行日志记录，然后调用超类中的方法。例如，如果在调用一个面板的 setBackground方法时出现了问题，就可以按照下面的方式，以匿名子类实例的形式创建一个代理对象：

```
JPanel panel = new
   JPanel()
   {
      public void setBackground(Color c)
      {
         Logger.global.info("setBackground: c=" + c);
         super.setBackground(c);
      }
   };
```

当调用setBackground方法时，就会产生一个日志消息。要想知道谁调用了这个方法，就要生成一个堆栈跟踪。

5）利用Throwable类提供的printStackTrace方法，可以从任何一个异常对象中获得堆栈情况。下面的代码将捕获任何异常，打印异常对象和堆栈跟踪，然后，重新抛出异常，以便能够找到相应的处理器。

```
try
{
   . . .
}
catch (Throwable t)
{
   t.printStackTrace();
   throw t;
}
```

不一定要通过捕获异常来生成堆栈跟踪。只要在代码的任何位置插入下面这条语句就可以获得堆栈跟踪：

```
Thread.dumpStack();
```

6）一般来说，堆栈跟踪显示在System.err上。也可以利用printStackTrace(PrintWriter s)方法将它发送到一个文件中。另外，如果想记录或显示堆栈跟踪，就可以采用下面的方式，将它

捕获到一个字符串中：

```
StringWriter out = new StringWriter();
new Throwable().printStackTrace(new PrintWriter(out));
String trace = out.toString();
```

有关PrintWriter 和StringWriter两个类的详细情况，请参看卷II第1章。

7）通常，将一个程序中的错误信息保存在一个文件中是非常有用的。然而，错误信息被发送到System.err中，而不是System.out中。因此，不能够通过运行下面的语句获取它们：

```
java MyProgram > errors.txt
```

而是采用下面的方式捕获错误流：

```
java MyProgram 2> errors.txt
```

要想在同一个文件中同时捕获System.err和System.out，需要使用下面这条命令

```
java MyProgram >& errors.txt
```

这条命令将工作在Windows外壳中。

8）让非捕获异常的堆栈跟踪出现在System.err中并不是一个很理想的方法。如果在客户端偶然看到这些消息，则会感到迷惑，并且在需要的时候也无法实现诊断目的。比较好的方式是将这些内容记录到一个文件中。在Java SE5.0中，可以调用静态的Thread.setDefaultUncaughtExceptionHandler方法改变非捕获异常的处理器：

```
Thread.setDefaultUncaughtExceptionHandler(
      new Thread.UncaughtExceptionHandler()
      {
         public void uncaughtException(Thread t, Throwable e)
         {
            save information in log file
         };
      });
```

9）要想观察类的加载过程，可以用-verbose标志启动Java虚拟机。这样就可以看到如下所示的输出结果：

```
[Opened /usr/local/jdk5.0/jre/lib/rt.jar]
[Opened /usr/local/jdk5.0/jre/lib/jsse.jar]
[Opened /usr/local/jdk5.0/jre/lib/jce.jar]
[Opened /usr/local/jdk5.0/jre/lib/charsets.jar]
[Loaded java.lang.Object from shared objects file]
[Loaded java.io.Serializable from shared objects file]
[Loaded java.lang.Comparable from shared objects file]
[Loaded java.lang.CharSequence from shared objects file]
[Loaded java.lang.String from shared objects file]
[Loaded java.lang.reflect.GenericDeclaration from shared objects file]
[Loaded java.lang.reflect.Type from shared objects file]
[Loaded java.lang.reflect.AnnotatedElement from shared objects file]
[Loaded java.lang.Class from shared objects file]
[Loaded java.lang.Cloneable from shared objects file]
...
```

有时候，这种方法有助于诊断由于类路径引发的问题。

10）如果曾经看到过Swing窗口，并对设计者能够采用某种管理手段将所有的组件排列整齐而感到好奇，就可以监视一下有关的信息。按下CTRL+SHIFT+F1，将会按照层次结构打印出所有组件的信息：

```
FontDialog[frame0,0,0,300x200,layout=java.awt.BorderLayout,...
  javax.swing.JRootPane[,4,23,292x173,layout=javax.swing.JRootPane$RootLayout,...
   javax.swing.JPanel[null.glassPane,0,0,292x173,hidden,layout=java.awt.FlowLayout,...
   javax.swing.JLayeredPane[null.layeredPane,0,0,292x173,...
    javax.swing.JPanel[null.contentPane,0,0,292x173,layout=java.awt.GridBagLayout,...
     javax.swing.JList[,0,0,73x152,alignmentX=null,alignmentY=null,...
      javax.swing.CellRendererPane[,0,0,0x0,hidden]
       javax.swing.DefaultListCellRenderer$UIResource[,-73,-19,0x0,...
     javax.swing.JCheckBox[,157,13,50x25,layout=javax.swing.OverlayLayout,...
     javax.swing.JCheckBox[,156,65,52x25,layout=javax.swing.OverlayLayout,...
     javax.swing.JLabel[,114,119,30x17,alignmentX=0.0,alignmentY=null,...
     javax.swing.JTextField[,186,117,105x21,alignmentX=null,alignmentY=null,...
     javax.swing.JTextField[,0,152,291x21,alignmentX=null,alignmentY=null,...
```

11）如果自定义Swing组件，却不能正确地显示，则会发现Swing图形调试器（Swing graphics debugger）是一个不错的工具。即使自己不编写组件类，亲眼看看组件的内容如何被绘制出来也非常有意义且有趣。要想对一个Swing组件进行调试，可以调用JComponent 类的setDebugGraphicsOptions方法。下面是调用这个方法的可选项：

DebugGraphics.FLASH_OPTION	在绘制每条线段、每个矩形、每个文本之前，用红色闪烁显示
DebugGraphics.LOG_OPTION	打印每次绘制操作的信息
DebugGraphics.BUFFERED_OPTION	显示在显示区域之外执行的缓冲操作
DebugGraphics.NONE_OPTION	关闭图形调试

我们已经发现：要使闪烁选项能够工作，就必须禁用“双重缓冲”。这是Swing为缓解更新窗口时屏幕抖动现象所采取的一种策略。下面是用来开启闪烁选项的代码：

```
RepaintManager.currentManager(getRootPane()).setDoubleBufferingEnabled(false);
((JComponent) getContentPane()).setDebugGraphicsOptions(DebugGraphics.FLASH_OPTION);
```

只要将这几行代码放置在框架构造器的尾部就可以了。当运行程序时，可以看到内容窗格以极慢的速度填充。如果要进行本地化调试，只要为单个组件调用setDebugGraphicsOptions方法即可。对于这种闪烁，可以设置的参数包括持续时间、次数和闪烁的颜色。有关DebugGraphics类更加详细的内容请参看在线文档。

12）Java SE 5.0增加了-Xlint选项，这样，编译器就可以对一些普遍容易出现的代码问题进行检查。例如，如果使用下面这条命令编译：

```
javac -Xlint:fallthrough
```

当switch语句中缺少break语句时，编译器就会给出报告（术语“lint”最初用来描述一种定位C程序中潜在问题的工具，现在通常用于描述查找可疑的、但不违背语法规则的代码问题的工具。）

下面列出了可以使用选项：

-Xlint 或-Xlint:all	执行所有的检查
-Xlint:deprecation	与-deprecation一样，检查反对使用的方法

```
-Xlint:fallthrough          检查switch语句中是否缺少break语句
-Xlint:finally              警告finally子句不能正常地执行
-Xlint:none                 不执行任何检查
-Xlint:path                 检查类路径和源代码路径上的所有目录是否存在
-Xlint:serial               警告没有serialVersionUID的串行化类（请参看卷II第1章）
-Xlint:unchecked            对通用类型与原始类型之间的危险转换给予警告（请参看第12章）
```

13）Java SE 5.0增加了对Java应用程序进行监控（monitoring）和管理（management）的支持。它允许利用虚拟机中的代理装置跟踪内存消耗、线程使用、类加载等情况。这个功能对于像应用程序服务器这样大型的、长时间运行的Java程序来说特别重要。下面是一个能够展示这种功能的例子：JDK加载了一个称为jconsole的图形工具，可以用于显示虚拟机性能的统计结果，如图11-3所示。找出运行虚拟机的操作系统进程的ID。在UNIX/Linux环境下，运行ps实用工具，在Windows环境下，使用任务管理器。然后运行jconsole程序：

```
jconsole processID
```

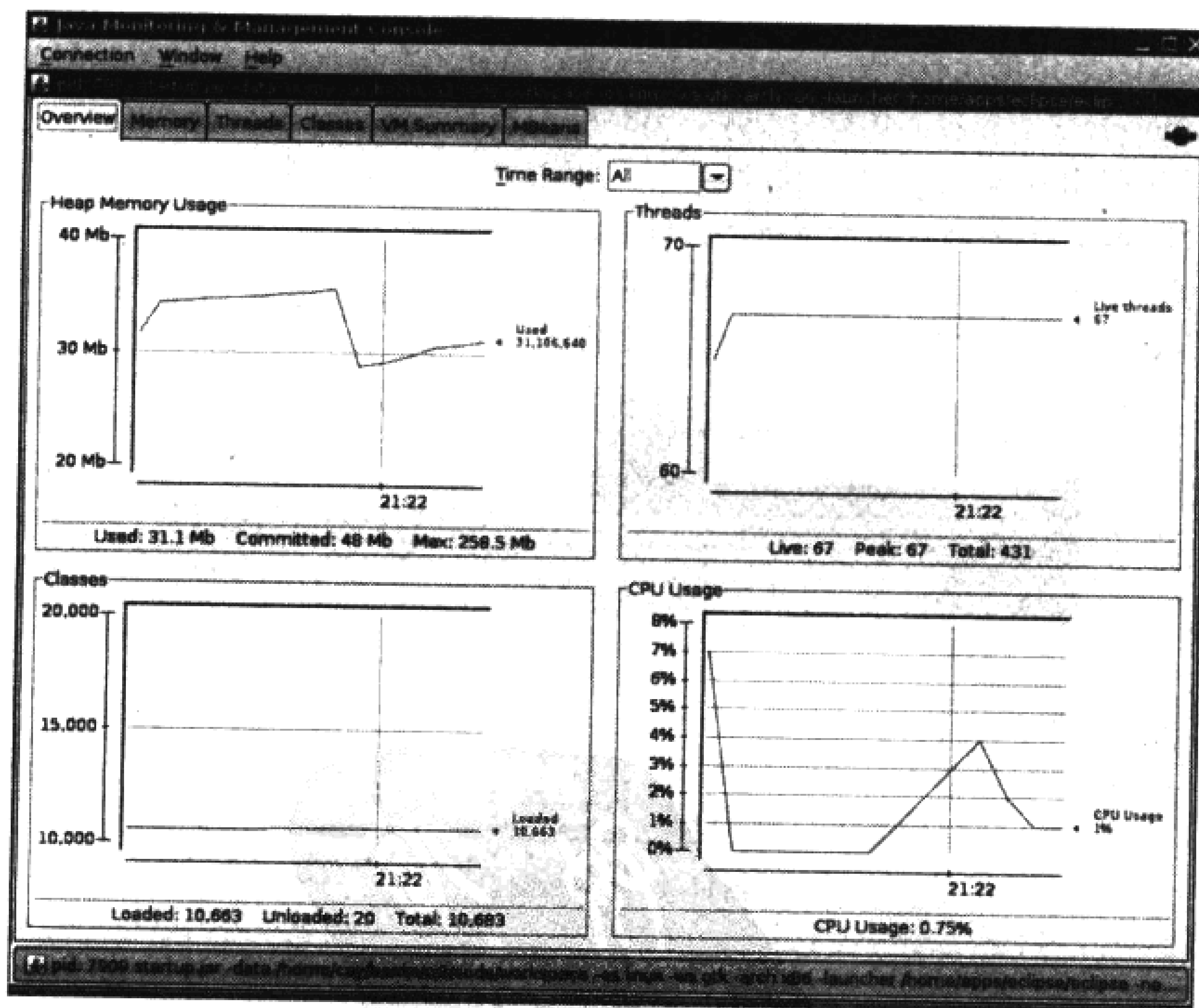
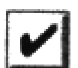


图11-3 jconsole程序

控制台给出了有关运行程序的大量信息。有关更加详细的信息参见http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html。

 **注释**：在Java SE 6之前，需要用-Dcom.sun.management.jmxremote option启动程序：

```
java -Dcom.sun.management.jmxremote MyProgram
jconsole processID
```

14）可以使用jmap实用工具获得一个堆的堆放处，其中显示了堆中的每个对象。使用命令如下：

```
jmap -dump:format=b,file=dumpFileName processID
jhat dumpFileName
```

然后，通过浏览器进入localhost:7000，将会运行一个网络应用程序，借此探查倾到对象时堆的内容。

15）如果使用-Xprof标志运行Java虚拟机，就会运行一个基本的剖析器来跟踪那些代码中经常被调用的方法。剖析信息将发送给System.out。输出结果中还会显示哪些方法是由just-in-time编译器编译的。

警告：*编译器的-X选项并没有正式支持，而且在有些JDK版本中并不存在这个选项。可以运行命令java -X得到所有非标准选项的列表。*

11.6.1 使用控制台窗口

在调试applet应用程序时可以在窗口中看到错误消息。在Java内置的配置面板中可以选择Show Java Console（请参看第10章）。Java Console窗口有一组滚动条，当消息已经滚动到窗口之外时，可以利用滚动条将它们滚动回来。用户将会发现：使用窗口明显优越于采用System.out和System.err将消息显示在DOS窗口中。

这里提供一个类似的窗口类，以便在调试程序时，能够在具有同样效果的窗口中查看调试消息。图11-4显示了运行ConsoleWindow类的效果。

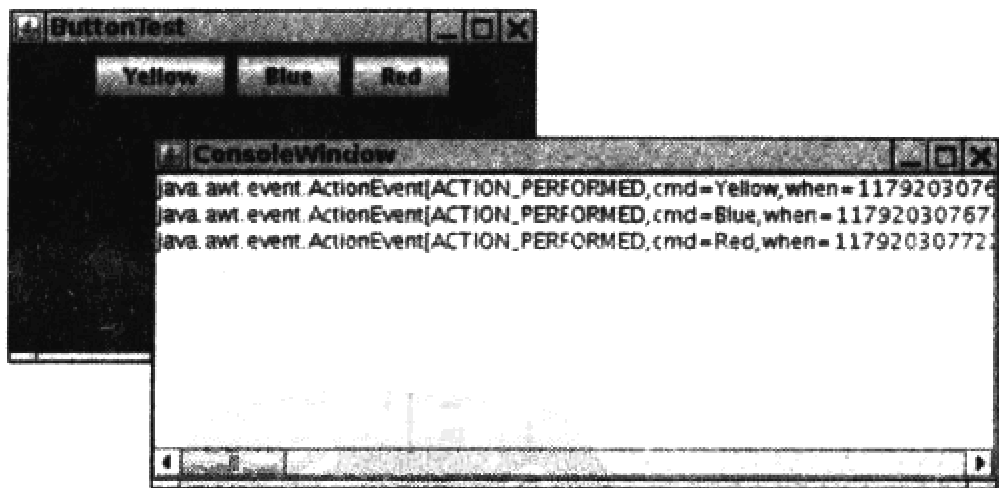


图11-4 控制台窗口

这个类的用法很简单。只需要进行下面这个调用就可以了：

```
ConsoleWindow.init()
```

然后，用常规的方式打印System.ouyt和System.err。

例11-3列出了ConsoleWindow类的代码清单。可以看到，这个类相当简单。消息将显示在JScrollPane内的JTextArea中。在这个类中，通过调用System.setOut和System.setErr方法将输出流和错误流定向到一个特定的流中，这个流将所有的消息加入到一个文本区中。

例11-3 ConsoleWindow.java

```
import javax.swing.*;
import java.io.*;

/**
   A window that displays the bytes sent to System.out and System.err
   @version 1.01 2004-05-10
   @author Cay Horstmann
*/
public class ConsoleWindow
{
   public static void init()
   {
      JFrame frame = new JFrame();
      frame.setTitle("ConsoleWindow");
      final JTextArea output = new JTextArea();
      output.setEditable(false);
      frame.add(new JScrollPane(output));
      frame.setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
      frame.setLocation(DEFAULT_LEFT, DEFAULT_TOP);
      frame.setFocusableWindowState(false);
      frame.setVisible(true);

      // define a PrintStream that sends its bytes to the output text area
      PrintStream consoleStream = new PrintStream(new
         OutputStream()
         {
            public void write(int b) {} // never called
            public void write(byte[] b, int off, int len)
            {
               output.append(new String(b, off, len));
            }
         });

      // set both System.out and System.err to that stream
      System.setOut(consoleStream);
      System.setErr(consoleStream);
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;
   public static final int DEFAULT_LEFT = 200;
   public static final int DEFAULT_TOP = 200;
}
```

11.6.2 跟踪AWT事件

当用Java设计一个奇特的用户界面时，需要知道AWT将什么事件发送到什么组件上。很遗憾，AWT文档对这部分内容讲述的轻描淡写。例如，假设当用户将鼠标移到屏幕的不同位置时，希望在状态栏上显示一条提示信息，此时应该捕获AWT产生的鼠标和焦点事件。

下面给出一个很有用的Eventtrace类，它可以监控这些事件，并打印输出所有事件的处理方法和参数。图11-5显示了被跟踪的事件。

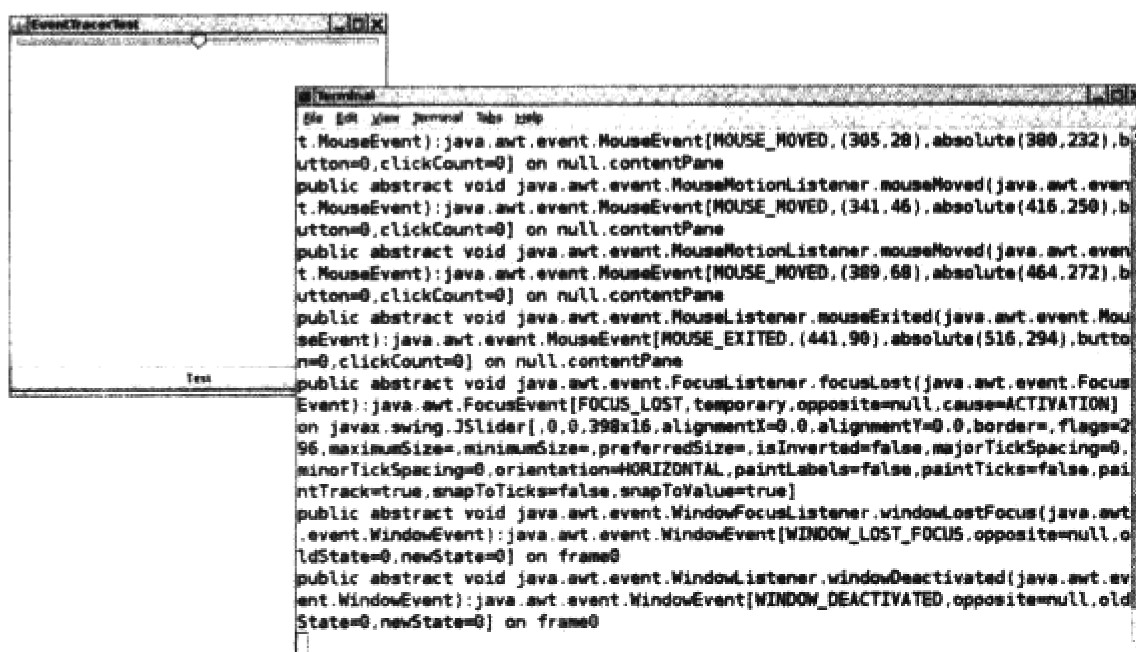


图11-5 The Eventtracer 类的运行结果

要想监测这些信息，需要希望跟踪事件的组件添加到事件跟踪器中：

```
EventTracer tracer = new EventTracer();
tracer.add(frame);
```

这样就会打印出所有事件的文字化描述信息，如下所示：

```
public abstract void java.awt.event.MouseListener.mouseExited(java.awt.event.MouseEvent):
java.awt.event.MouseEvent[MOUSE_EXITED,(408,14),button=0,clickCount=0] on javax.swing.JBut-
ton[,0,345,400x25,...]
public abstract void java.awt.event.FocusListener.focusLost(java.awt.event.FocusEvent):
java.awt.event.FocusEvent[FOCUS_LOST,temporary,opposite=null] on javax.swing.JButton[,0,345,400x25,...]
```

可能还希望将这些信息输出到一个文件或控制台窗口上。前面已经讲过实现的具体方式。例11-4是Eventtracer类的程序代码。尽管这个类实现起来非常复杂，但其思路却十分简单。

例11-4 EventTracer.java

```
import java.awt.*;
import java.beans.*;
import java.lang.reflect.*;

/**
 * @version 1.31 2004-05-10
 * @author Cay Horstmann
 */
public class EventTracer
{
   public EventTracer()
   {
      // the handler for all event proxies
      handler = new InvocationHandler()
         {
            public Object invoke(Object proxy, Method method, Object[] args)
            {
               System.out.println(method + ":" + args[0]);
               return null;
            }
         };
   }
```

```

   /**
    * Adds event tracers for all events to which this component and its children can listen
    * @param c a component
    */
   public void add(Component c)
   {
      try
      {
         // get all events to which this component can listen
         BeanInfo info = Introspector.getBeanInfo(c.getClass());

         EventSetDescriptor[] eventSets = info.getEventSetDescriptors();
         for (EventSetDescriptor eventSet : eventSets)
            addListener(c, eventSet);
      }
      catch (IntrospectionException e)
      {
      }
      // ok not to add listeners if exception is thrown

      if (c instanceof Container)
      {
         // get all children and call add recursively
         for (Component comp : ((Container) c).getComponents())
            add(comp);
      }
   }

   /**
    * Add a listener to the given event set
    * @param c a component
    * @param eventSet a descriptor of a listener interface
    */
   public void addListener(Component c, EventSetDescriptor eventSet)
   {
      // make proxy object for this listener type and route all calls to the handler
      Object proxy = Proxy.newProxyInstance(null, new Class[] { eventSet.getListenerType() },
            handler);

      // add the proxy as a listener to the component
      Method addListenerMethod = eventSet.getAddListenerMethod();
      try
      {
         addListenerMethod.invoke(c, proxy);
      }
      catch (InvocationTargetException e)
      {
      }
      catch (IllegalAccessException e)
      {
      }
      // ok not to add listener if exception is thrown
   }

   private InvocationHandler handler;
}
```

1）当用add方法将一个组件添加到事件跟踪器时，JavaBeans自省类就会分析这个组件中形如addXxxListener(XxxEvent) 的所有方法（有关JavaBeans的详细介绍请看卷II的第8章）。对于每一个匹配的方法，都会生成一个EventSetDescriptor对象，并将它传递给addListener方法。

2）如果组件是一个容器，就列举其中的每一个组件，并为每个组件递归地调用add方法。

3）调用addListener方法需要提供两个参数：一个是希望监测其事件的组件；另一个是事件设置描述符。调用EventSetDescriptor 类的getListenerType方法可以返回一个Class对象，这个对象描述了诸如 ActionListener或ChangeListener这样的事件监听器接口。随后，要为接口创建一个代理对象。这个代理处理器只打印被调用的事件方法和参数。调用EventSetDescriptor类的getAddListenerMethod方法可以返回一个Method对象，使用这个对象可以增加一个代理对象来作为组件的事件监听器。

这个程序是体现反射机制的一个很好例子。在这里，并不需要一定为JButton类设计一个addActionListener方法，为JSlider类再设计一个 addChangeListener方法。反射机制会发现这些信息。

例11-5测试了事件跟踪器。这个程序显示了一个含有一个按钮，一个滚动条的窗口框架，并且能够跟踪这些组件生成的事件。

例11-5 EventTracerTest.java

```
import java.awt.*;

import javax.swing.*;

/**
 * @version 1.13 2007-06-12
 * @author Cay Horstmann
 */
public class EventTracerTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new EventTracerFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

class EventTracerFrame extends JFrame
{
   public EventTracerFrame()
   {
      setTitle("EventTracerTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

```

```
      // add a slider and a button
      add(new JSlider(), BorderLayout.NORTH);
      add(new JButton("Test"), BorderLayout.SOUTH);

      // trap all events of components inside the frame
      EventTracer tracer = new EventTracer();
      tracer.add(this);
   }

   public static final int DEFAULT_WIDTH = 400;
   public static final int DEFAULT_HEIGHT = 400;
}
```

11.6.3 AWT的Robot类

Java SE1.3版本增加了一个Robot类，它用来将敲击键盘和点击鼠标的事件发送给AWT程序，并能够对用户界面进行自动地检测。

要想获得Robot对象，首先要得到一个GraphicsDevice对象。通过下面的一系列调用获得默认的屏幕设备：

```
GraphicsEnvironment environment = GraphicsEnvironment.getLocalGraphicsEnvironment();
GraphicsDevice screen = environment.getDefaultScreenDevice();
```

然后构造一个Robot对象：

```
Robot robot = new Robot(screen);
```

要想发送一个敲击键盘的事件，就需要告诉robot模拟按下键和释放键的动作：

```
robot.keyPress(KeyEvent.VK_TAB);
robot.keyRelease(KeyEvent.VK_TAB);
```

要想发送一个点击鼠标的事件，首先要模拟移动鼠标的事件，然后再模拟按下和释放鼠标的事件：

```
robot.mouseMove(x, y); // x and y are absolute screen pixel coordinates.
robot.mousePress(InputEvent.BUTTON1_MASK);
robot.mouseRelease(InputEvent.BUTTON1_MASK);
```

下面的思路是：模拟键盘和鼠标的输入，然后进行屏幕快照，以便查看应用程序是否实施了预期的操作。可以调用createScreenCapture方法实现捕捉屏幕的操作：

```
Rectangle rect = new Rectangle(x, y, width, height);
BufferedImage image = robot.createScreenCapture(rect);
```

矩形坐标使用的也是绝对像素坐标。

最后，通常希望在上面的各条命令之间加上一个短暂的延迟，以保证应用程序能够捕获到各个事件。设置延迟的方法是：调用delay方法并传递一个以毫秒为单位的延迟时间，例如：

```
robot.delay(1000); // delay by 1000 milliseconds
```

例11-6的程序说明了如何使用robot的方法。Rrobot检测了在第8章中已经出现过的按钮测试程序。首先，按下一个空格键来激活最左侧的按钮，然后robot等待2秒，以便能够看到它所做的一切操作。等待一会儿之后，robot将模拟按下TAB键和空格键来点击下一个按钮。最后，

模拟用鼠标点击第三个按钮（可能需要调整程序中的x、y坐标，以保证能够正确地按下按钮）。这个程序最后得到了屏幕截图，并将它显示在另一个框架中，如图11-6所示。

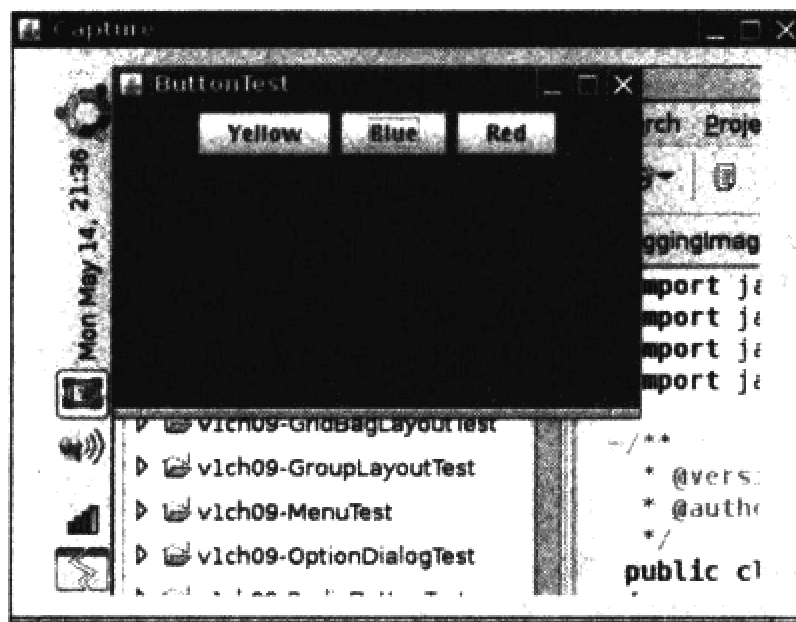


图11-6 用AWT的robot捕捉屏幕

从这个示例可以看到，Robot类自身并不适用于进行用户界面的测试。实际上，它是用于构建基本测试工具的基础构件。一个专业的测试工具应该具有捕获、存储和再现用户交互场景的功能，并能够确定组件在屏幕中的位置，以调整鼠标点击位置。作为日趋流行的Java应用程序，我们期盼能够出现更加成熟的测试工具。

例11-6 RobotTest.java

```
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import javax.swing.*;

/**
 * @version 1.03 2007-06-12
 * @author Cay Horstmann
 */
public class RobotTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               // make frame with a button panel

               ButtonFrame frame = new ButtonFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);

               // attach a robot to the screen device

               GraphicsEnvironment environment = GraphicsEnvironment.getLocalGraphicsEnvironment();
```

```
               GraphicsDevice screen = environment.getDefaultScreenDevice();

               try
               {
                  Robot robot = new Robot(screen);
                  runTest(robot);
               }
               catch (AWTException e)
               {
                  e.printStackTrace();
               }
            }
         });
   }

   /**
    * Runs a sample test procedure
    * @param robot the robot attached to the screen device
    */
   public static void runTest(Robot robot)
   {
      // simulate a space bar press
      robot.keyPress(' ');
      robot.keyRelease(' ');

      // simulate a tab key followed by a space
      robot.delay(2000);
      robot.keyPress(KeyEvent.VK_TAB);
      robot.keyRelease(KeyEvent.VK_TAB);
      robot.keyPress(' ');
      robot.keyRelease(' ');

      // simulate a mouse click over the rightmost button
      robot.delay(2000);
      robot.mouseMove(200, 50);
      robot.mousePress(InputEvent.BUTTON1_MASK);
      robot.mouseRelease(InputEvent.BUTTON1_MASK);

      // capture the screen and show the resulting image
      robot.delay(2000);
      BufferedImage image = robot.createScreenCapture(new Rectangle(0, 0, 400, 300));

      ImageFrame frame = new ImageFrame(image);
      frame.setVisible(true);
   }
}

/**
 * A frame to display a captured image
 */
class ImageFrame extends JFrame
{
   /**
    * @param image the image to display
    */
   public ImageFrame(Image image)
   {
```

```
        setTitle("Capture");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        JLabel label = new JLabel(new ImageIcon(image));
        add(label);
    }

    public static final int DEFAULT_WIDTH = 450;
    public static final int DEFAULT_HEIGHT = 350;
}
```

API java.awt.GraphicsEnvironment 1.2

- `static GraphicsEnvironment getLocalGraphicsEnvironment()`

  返回本地的图形环境。

- `GraphicsDevice getDefaultScreenDevice()`

  返回默认的屏幕设备。需要注意，使用多台监视器的计算机，每一个屏幕有一个图形设备。通过调用 getScreenDevices方法可以得到一个保存所有屏幕设备的数组。

API java.awt.Robot 1.3

- `Robot(GraphicsDevice device)`

  构造一个能够与给定设备交互的robot对象。

- `void keyPress(int key)`
- `void keyRelease(int key)`

  模拟按下或释放按键。

  参数：key　　键码。有关键码更加详细的信息请参看KeyStroke类

- `void mouseMove(int x, int y)`

  模拟移动鼠标。

  参数：x, y　　用绝对像素坐标表示的鼠标位置

- `void mousePress(int eventMask)`
- `void mouseRelease(int eventMask)`

  模拟按下或释放鼠标键。

  参数：eventMask　描述鼠标键的事件掩码。有关事件掩码更加详细的信息请参看InputEvent

- `void delay(int milliseconds)`

  根据给定毫秒数延迟robot。

- `BufferedImage createScreenCapture(Rectangle rect)`

  截取屏幕的一部分。

  参数：rect　　用绝对像素坐标表示的所截取的矩形。

11.7 使用调试器

借助于打印语句进行调试并不是一件令人愉快的事情。在这个过程中，需要频繁地增加和删除这些语句，然后，再对程序进行重新编译。使用调试器情况就好多了。调试器会全速运行程序，直到遇到一个断点为止，然后可以查看任何感兴趣的内容。

在这里，有意识地对第8章中的ButtonTest程序进行了一些破坏性的修改，例11-7是Bitton Test程序的错误版本。当点击任何一个按钮时，没有发生任何事情。请看一下源代码，程序原本设计的功能是：点击一下按钮，将背景颜色设置为与按钮名相同的颜色。

例11-7 BuggyButtonTest.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * @version 1.22 2007-05-14
 * @author Cay Horstmann
 */
public class BuggyButtonTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               BuggyButtonFrame frame = new BuggyButtonFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

class BuggyButtonFrame extends JFrame
{
   public BuggyButtonFrame()
   {
      setTitle("BuggyButtonTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // add panel to frame

      BuggyButtonPanel panel = new BuggyButtonPanel();
      add(panel);
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;
}

class BuggyButtonPanel extends JPanel
```

```java
{
   public BuggyButtonPanel()
   {
      ActionListener listener = new ButtonListener();

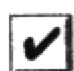
      JButton yellowButton = new JButton("Yellow");
      add(yellowButton);
      yellowButton.addActionListener(listener);

      JButton blueButton = new JButton("Blue");
      add(blueButton);
      blueButton.addActionListener(listener);

      JButton redButton = new JButton("Red");
      add(redButton);
      redButton.addActionListener(listener);
   }

   private class ButtonListener implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         String arg = event.getActionCommand();
         if (arg.equals("yellow")) setBackground(Color.yellow);
         else if (arg.equals("blue")) setBackground(Color.blue);
         else if (arg.equals("red")) setBackground(Color.red);
      }
   }
}
```

在这样一个简短的程序中，通过阅读源代码可以发现bug。假设这是一个非常复杂的程序，要想通过阅读程序查找错误就不太切合实际了。下面将介绍一下如何使用Eclipse调试器定位错误。



注释：如果使用像JSwat（http://www.bluemarsh.com/java/jswat/）这样的独立调试器，或者非常陈旧的jdb，那么就首先必须用-g选项编译程序。例如：

```
javac -g BuggyButtonTest.java
```

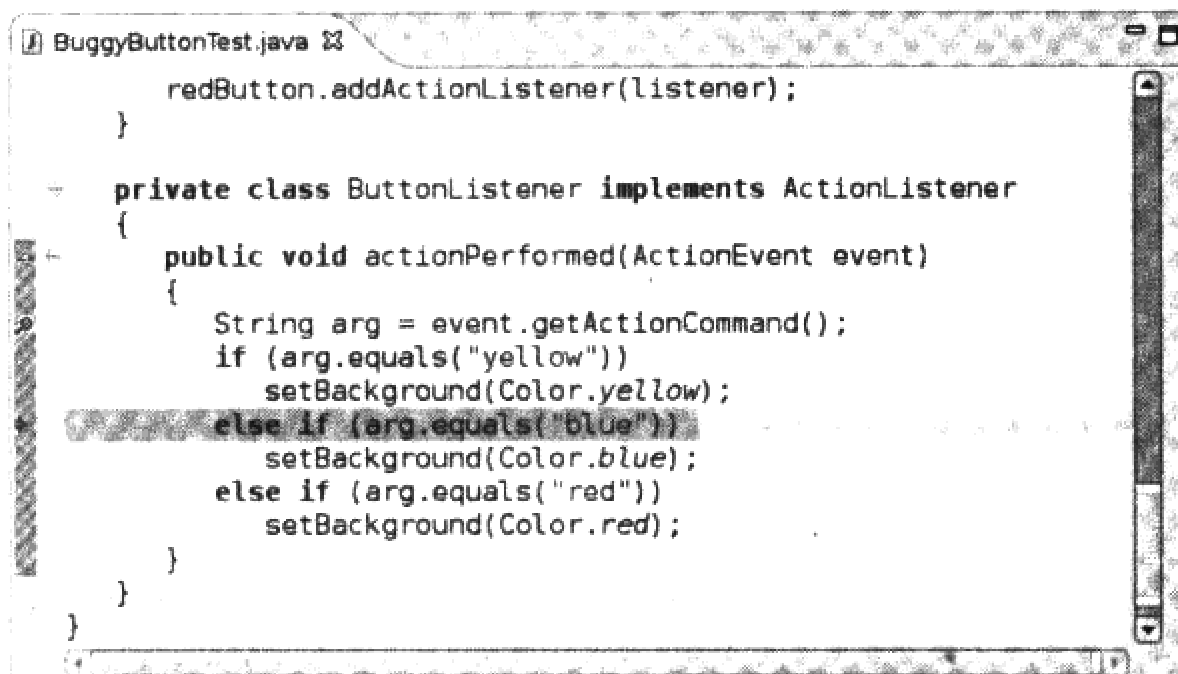
在集成环境中，这个操作是自动完成的。

在Eclipse中，可以选择菜单Run→Debug As→Java Application启动调试器。此时，程序就开始运行。

在actionPerformed方法的第一行中设置一个断点：将鼠标移到希望设置断点的那一行，然后在左侧的空白处点击鼠标右键，并选择Toggle Breakpoint。

一旦Java开始处理actionPerformed方法中的代码就会遇到断点。为此，点击Yellow按钮。调试器将在actionPerformed方法的开始处中断。如图11-7所示。

有两个单步跟踪应用程序的命令。“Step Into”命令跟踪到每个方法调用的内部。“Step Over”命令定位到下一行，而并不跟踪到方法调用的内部。Eclipse使用菜单选项Run→Step Into和Run→Step Over，键盘快捷键示F5和F6。发出两次“Step Over”命令，看一下刚表的位置。



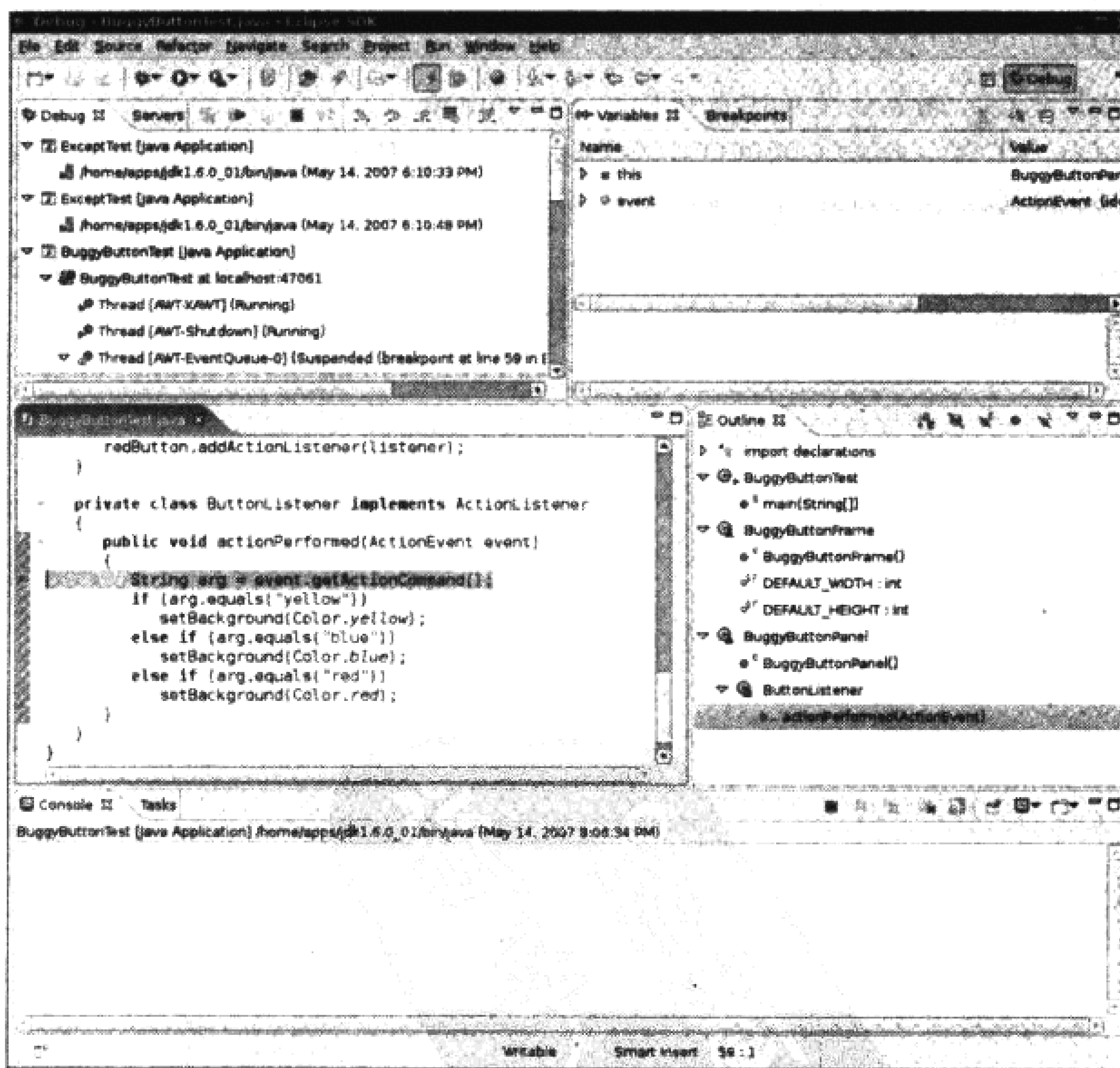
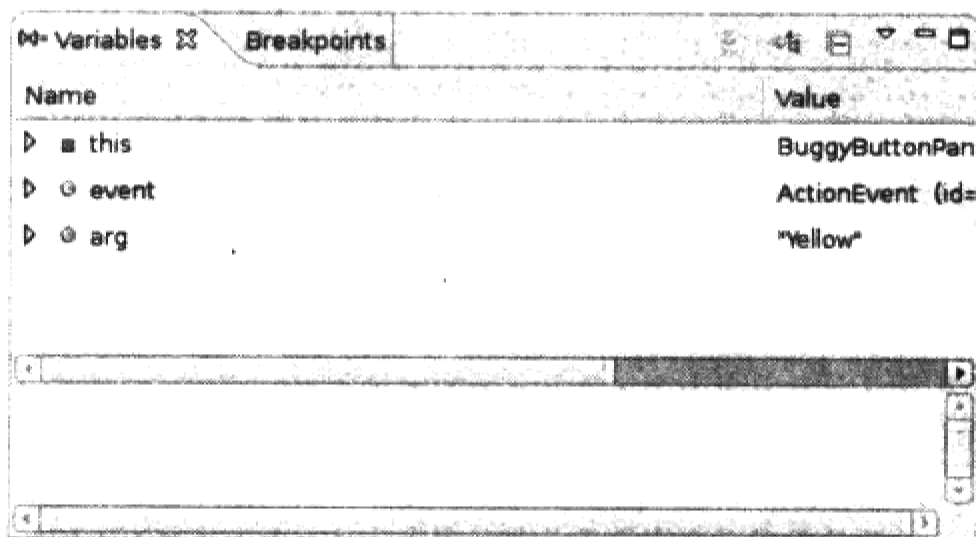


图11-7 停止在断点处

这不是应该发生的事情。假设程序调用setColor(Color.yellow)，然后退出方法。查看局部变量，并检查arg变量的值：



现在，可以看到所发生的变化。arg的值是“Yellow”，Y是大写字母，但是，下面比较测试中的y是小写字母：

```
if (arg.equals("yellow"))
```

神秘般地解决了。

要想退出调试器，需要从菜单中选择Run→Terminate。

在Eclipse中，还有一些高级的调试命令，但是，大都使用刚才看到的简单技术。其他的调试器，如NetBeans调试器页都提供了类似的命令。

本章介绍了异常处理和测试于调试遇到的一些技术。接下来的两章将介绍范型程序设计以及最重要的应用：Java集合框架。

第12章 泛型程序设计

▲ 为什么要使用泛型程序设计
▲ 简单泛型类的定义
▲ 泛型方法
▲ 类型变量的限定
▲ 泛型代码和虚拟机
▲ 约束与局限性
▲ 泛型类型的继承规则
▲ 通配符类型
▲ 反射和泛型

从Java程序设计语言1.0版发布以来，变化最大的部分就是泛型。致使Java SE 5.0中增加泛型机制的主要原因是为了满足在1999年制定的最早的Java规范需求之一（JSR 14）。专家组花费了5年左右的时间用来定义规范和测试实现。

泛型正是我们需要的程序设计手段。使用泛型机制编写的程序代码要比那些杂乱地使用Object变量，然后再进行强制类型转换的代码具有更好的安全性和可读性。泛型对于集合类尤其有用，例如，ArrayList就是一个无处不有的集合类。

至少在表面上看来，泛型很像C++中的模板。与Java一样，在C++中，模板也是最先被添加到语言中支持强类型集合的。但是，多年之后人们发现模板还有其他的用武之地。学习完本章的内容可以发现Java中的泛型在程序中也是如此。

12.1 为什么要使用泛型程序设计

泛型程序设计（Generic programming）意味着编写的代码可以被很多不同类型的对象所重用。例如，我们并不希望为聚集String和File对象分别设计不同的类。实际上，也不需要这样做，因为一个ArrayList类可以聚集任何类型的对象。这是一个泛型程序设计的实例。

在Java SE 5.0之前，Java泛型程序设计是用继承实现的。ArrayList类只维护一个Object引用的数组：

```
public class ArrayList // before Java SE 5.0
{
   public Object get(int i) { . . . }
   public void add(Object o) { . . . }
   . . .
   private Object[] elementData;
}
```

这样的实现有两个问题。当获取一个值时必须进行强制类型转换。

```
ArrayList files = new ArrayList();
. . .
String filename = (String) names.get(0);
```

此外，这里没有错误检查。可以向数组列表中添加任何类的对象。

```
files.add(new File(". . ."));
```

对于这个调用，编译和运行都不会出错。然而在其他地方，如果将get的结果强制类型转换为String类型，就会产生一个错误。

泛型提供了一个更好的解决方案：*类型参数*（type parameters）。ArrayList类有一个类型参数用来指示元素的类型：

```
ArrayList<String> files = new ArrayList<String>();
```

这使得代码具有更好的可读性。人们一看就知道这个数组列表中包含的是String对象。

编译器也可以很好地利用这个信息。当调用get的时候，不需要进行强制类型转换，编译器就知道返回值类型为String，而不是Object：

```
String filename = files.get(0);
```

编译器还知道ArrayList<String>中add方法有一个类型为String的参数。这将比使用Object类型的参数安全一些。现在，编译器可以进行检查，避免插入错误类型的对象。例如：

```
files.add(new File(". . .")); // can only add String objects to an ArrayList<String>
```

是无法通过编译的。出现编译错误比类在运行时出现类的强制类型转换异常要好得多。类型参数的魅力在于：使得程序具有更好的可读性和安全性。

谁想成为泛型程序员？

使用像ArrayList的泛型类很容易。大多数Java程序员都使用ArrayList<String>这样的类型，就好像它们已经构建在语言之中，像String[]数组一样。（当然，数组列表比数组要好一些，因为它可以自动扩展。）

但是，实现一个泛型类并没有那么容易。对于类型参数，使用这段代码的程序员可能想要内置（plug in）所有的类。他们希望在没有过多的限制以及混乱的错误消息的状态下，做所有的事情。因此，一个泛型程序员的任务就是预测出所用类的未来可能有的所有用途。

这一任务难到什么程度呢？下面是标准类库的设计者们肯定产生争议的一个典型问题。ArrayList类有一个方法addAll用来添加另一个集合的全部元素。程序员可能想要将ArrayList<Manager>中的所有元素添加到ArrayList<Employee>中去。然而，反过来就不行了。如果只能允许前一个调用，而不能允许后一个调用呢？Java语言的设计者发明了一个具有独创性的新概念，*通配符类型*（wildcard type），它解决了这个问题。通配符类型非常抽象，然而，它们能让库的构建者编写出尽可能灵活的方法。

泛型程序设计划分为3个熟练级别。基本级别是，仅仅使用泛型类——典型的是像ArrayList这样的集合——不必考虑它们的工作方式与原因。大多数应用程序员将会停留在这一级别上，直到出现了什么问题。当把不同的泛型类混合在一起时，或是在与对类型参数一无所知的遗留的代码进行衔接时，可能会看到含混不清的错误消息。如果这样的话，就需要学习Java泛型来系统地解决这些问题，而不要胡乱地猜测。当然，最终可能想要实现自己的泛型类与泛型方法。

应用程序员很可能不喜欢编写太多的泛型代码。Sun公司的人已经做出了很大的努力，为

所有的集合类提供了类型参数。凭经验来说，那些原本涉及许多来自通用类型（如Object或Comparable接口）的强制类型转换的代码一定会因使用类型参数而受益。

本章介绍实现自己的泛型代码需要了解的各种知识。希望大多数读者可以利用这些知识解决一些疑难问题，并满足对于参数化集合类的内部工作方式的好奇心。

12.2 简单泛型类的定义

一个泛型类（generic class）就是具有一个或多个类型变量的类。本章使用一个简单的Pair类作为例子。对于这个类来说，我们只关注泛型，而不会为数据存储的细节烦恼。下面是Pair类的代码：

```
public class Pair<T>
{
   public Pair() { first = null; second = null; }
   public Pair(T first, T second) { this.first = first;  this.second = second; }

   public T getFirst() { return first; }
   public T getSecond() { return second; }

   public void setFirst(T newValue) { first = newValue; }
   public void setSecond(T newValue) { second = newValue; }

   private T first;
   private T second;
}
```

Pair类引入了一个类型变量T，用尖括号（< >）括起来，并放在类名的后面。泛型类可以有多个类型变量。例如，可以定义Pair类，其中第一个域和第二个域使用不同的类型：

```
public class Pair<T, U> { . . . }
```

类定义中的类型变量指定方法的返回类型以及域和局部变量的类型。例如，

```
private T first; // uses type variable
```

✔ **注释**：类型变量使用大写形式，且比较短，这是很常见的。在Java库中，使用变量E表示集合的元素类型，K和V分别表示表的关键字与值的类型。T（需要时还可以用临近的字母U和S）表示"任意类型"。

用具体的类型替换类型变量就可以实例化泛型类型，例如：

```
Pair<String>
```

可以将结果想像成带有构造器的普通类：

```
Pair<String>()
Pair<String>(String, String)
```

和方法：

```
String getFirst()
String getSecond()
void setFirst(String)
void setSecond(String)
```

换句话说，泛型类可看作普通类的工厂。

例12-1中的程序使用了Pair类。静态的minmax方法遍历了数组并同时计算出最小值和最大值。它用一个Pair对象返回了两个结果。回想一下compareTo方法比较两个字符串，如果字符串相同则返回0；如果按照字典顺序，第一个字符串比第二个字符串靠前，就返回负值，否则，返回正值。



C++注释：从表面上看，Java的泛型类类似于C++的模板类。惟一明显的不同是Java没有专用的template关键字。但是，在本章中读者将会看到，这两种机制有着本质的区别。

例12-1 PairTest1.java

```
/**
 * @version 1.00 2004-05-10
 * @author Cay Horstmann
 */
public class PairTest1
{
   public static void main(String[] args)
   {
      String[] words = { "Mary", "had", "a", "little", "lamb" };
      Pair<String> mm = ArrayAlg.minmax(words);
      System.out.println("min = " + mm.getFirst());
      System.out.println("max = " + mm.getSecond());
   }
}

class ArrayAlg
{
   /**
    * Gets the minimum and maximum of an array of strings.
    * @param a an array of strings
    * @return a pair with the min and max value, or null if a is null or empty
    */
   public static Pair<String> minmax(String[] a)
   {
      if (a == null || a.length == 0) return null;
      String min = a[0];
      String max = a[0];
      for (int i = 1; i < a.length; i++)
      {
         if (min.compareTo(a[i]) > 0) min = a[i];
         if (max.compareTo(a[i]) < 0) max = a[i];
      }
      return new Pair<String>(min, max);
   }
}
```

12.3 泛型方法

前面已经介绍了如何定义一个泛型类。实际上，还可以定义一个带有类型参数的简单方法。

```
class ArrayAlg
{
```

```
    public static <T> T getMiddle(T[] a)
    {
        return a[a.length / 2];
    }
}
```

这个方法是在普通类中定义的，而不是在泛型类中定义的。然而，这是一个泛型方法，可以从尖括号和类型变量看出这一点。注意，类型变量放在修饰符（这里是public static）的后面，返回类型的前面。

泛型方法可以定义在普通类中，也可以定义在泛型类中。

当调用一个泛型方法时，在方法名前的尖括号中放入具体的类型：

```
String[] names = { "John", "Q.", "Public" };
String middle = ArrayAlg.<String>getMiddle(names);
```

在这种情况（实际也是大多数情况）下，方法调用中可以省略 <String> 类型参数。编译器有足够的信息能够推断出所调用的方法。它用names的类型（即String[ ]）与泛型类型T[ ] 进行匹配并推断出T一定是String。也就是说，可以调用

```
String middle = ArrayAlg.getMiddle(names);
```

在大多数情况下，对于泛型方法的类型引用没有问题。偶尔，编译器也会提示错误，此时需要解译错误报告。看一看下面这个示例：

```
double middle = ArrayAlg.getMiddle(3.14, 1729, 0);
```

错误消息是："found:java.lang.Number&java.lang.Comparable<? extends java.lang.Number&java.lang.Comparable<?>>, required:double"。在本章稍后将学习如何解译"found"类型的声明。简单地说，编译器将会自动打包参数为1个Double和2个Integer对象，而后寻找这些类的共同超类型。事实上，找到2个这样的超类型：Number和Comparable接口，其本身也是一个泛型类型。在这种情况下，可以采取的补救措施是将所有的参数写为double值。

提示：如果想知道编译器对一个泛型方法调用最终推断出哪种类型，Peter von der Ahé推荐了这样一个窍门：有目的地引入一个错误，并研究所产生的错误消息。例如，看一下调用ArrayAlg.getMiddle("Hello",0,null)。将结果赋给JButton，这不可能是正确的。将会得到一个错误报告"found: java.lang.Object&java.io.Serializable&java.lang.Comparable<? extends java.lang. Object&java.io.Serializable&java.lang.Comparable<?>>."。大致的意思是：可以将结果赋给Object、Serialiable或Comparable。

C++注释：在C++中将类型参数放在方法名后面，有可能会导致语法分析的歧义。例如，g(f<a, b>(c)) 可以理解为"用f<a, b>(c) 的结果调用g"，或者理解为"用两个布尔值f<a和b>(c) 调用g"。

12.4 类型变量的限定

有时，类或方法需要对类型变量加以约束。下面是一个典型的例子。我们要计算数组中的最小元素：

```
class ArrayAlg
{
   public static <T> T min(T[] a) // almost correct
   {
      if (a == null || a.length == 0) return null;
      T smallest = a[0];
      for (int i = 1; i < a.length; i++)
         if (smallest.compareTo(a[i]) > 0) smallest = a[i];
      return smallest;
   }
}
```

但是，这里有一个问题。请看一下min方法的代码内部。变量smallest类型为T，这意味着它可以是任何一个类的对象。怎么才能确信T所属的类有compareTo方法呢？

解决这个问题的方案是将T限制为实现了Comparable接口（只含一个方法compareTo的标准接口）的类。可以通过对类型变量T设置*限定*（bound）实现这一点：

```
public static <T extends Comparable> T min(T[] a) . . .
```

实际上Comparable接口本身就是一个泛型类型。目前，我们忽略其复杂性以及编译器产生的警告。第12.8节讨论了如何在Comparable接口中适当地使用类型参数。

现在，泛型的min方法只能被实现了Comparable接口的类（如String、Date等）的数组调用。由于Rectangle类没有实现Comparable接口，所以调用min将会产生一个编译错误。

C++注释：*在C++中不能对模板参数的类型加以限制。如果程序员用一个不适当的类型实例化一个模板，将会在模板代码中报告一个（通常是含糊不清的）错误消息。*

读者或许会感到奇怪——在此为什么使用关键字extends而不是implements？毕竟，Comparable是一个接口。下面的符号

```
<T extends Bounding Type >
```

表示T应该是绑定类型的*子类型*（subtype）。T和绑定类型可以是类，也可以是接口。选择关键字extends的原因是更接近子类的概念，并且Java的设计者也不打算在语言中再添加一个新的关键字（如sub）。

一个类型变量或通配符可以有多个限定，例如

```
T extends Comparable & Serializable
```

限定类型用“&”分隔，而逗号用来分隔类型变量。

在Java的继承中，可以根据需要拥有多个接口超类型，但限定中至多有一个类。如果用一个类作为限定，它必须是限定列表中的第一个。

在例12-2的程序中，重新编写了一个泛型方法minmax。这个方法计算泛型数组的最大值和最小值，并返回Pair<T>。

例12-2 PairTest2.java

```
import java.util.*;

/**
```

```
 * @version 1.00 2004-05-10
 * @author Cay Horstmann
 */
public class PairTest2
{
   public static void main(String[] args)
   {
      GregorianCalendar[] birthdays =
         {
            new GregorianCalendar(1906, Calendar.DECEMBER, 9), // G. Hopper
            new GregorianCalendar(1815, Calendar.DECEMBER, 10), // A. Lovelace
            new GregorianCalendar(1903, Calendar.DECEMBER, 3), // J. von Neumann
            new GregorianCalendar(1910, Calendar.JUNE, 22), // K. Zuse
         };
      Pair<GregorianCalendar> mm = ArrayAlg.minmax(birthdays);
      System.out.println("min = " + mm.getFirst().getTime());
      System.out.println("max = " + mm.getSecond().getTime());
   }
}

class ArrayAlg
{
   /**
      Gets the minimum and maximum of an array of objects of type T.
      @param a an array of objects of type T
      @return a pair with the min and max value, or null if a is
      null or empty
   */
   public static <T extends Comparable> Pair<T> minmax(T[] a)
   {
      if (a == null || a.length == 0) return null;
      T min = a[0];
      T max = a[0];
      for (int i = 1; i < a.length; i++)
      {
         if (min.compareTo(a[i]) > 0) min = a[i];
         if (max.compareTo(a[i]) < 0) max = a[i];
      }
      return new Pair<T>(min, max);
   }
}
```

12.5 泛型代码和虚拟机

虚拟机没有泛型类型对象——所有对象都属于普通类。在泛型实现的早期版本中，甚至能够将使用泛型的程序编译为在1.0虚拟机上运行的类文件！这个向后兼容性在Java泛型开发的后期被放弃了。如果使用Sun的编译器来编译使用Java泛型的代码，结果类文件将不能在5.0之前的虚拟机上运行。



注释：如果想拥有泛型的优势，同时又保留旧的虚拟机字节码兼容性，请查阅：http://sourceforge.net/ projects/retroweaver。Retroweaver程序重写类文件以便与旧的虚拟机兼容。

无论何时定义一个泛型类型，都自动提供了一个相应的*原始类型*（raw type）。原始类型的名字就是删去类型参数后的泛型类型名。*擦除*（erased）类型变量，并替换为限定类型（无限定的变量用Object）。

例如，Pair<T>的原始类型如下所示：

```
public class Pair
{
   public Pair(Object first, Object second)
   {
      this.first = first;
      this.second = second;
   }

   public Object getFirst() { return first; }
   public Object getSecond() { return second; }

   public void setFirst(Object newValue) { first = newValue; }
   public void setSecond(Object newValue) { second = newValue; }

   private Object first;
   private Object second;
}
```

因为T是一个无限定的变量，所以直接用Object替换。

结果是一个普通的类，就好像泛型引入Java语言之前已经实现的那样。

在程序中可以包含不同类型的Pair，例如，Pair<String> 或 Pair<GregorianCalendar>。而擦除类型后就变成原始的Pair类型了。

C++注释：*就这点而言，Java泛型与C++模板有很大的区别。C++中每个模板的实例化产生不同的类型，这一现象称为“模板代码膨胀”。Java不存在这个问题的困扰。*

原始类型用第一个限定的类型变量来替换，如果没有给定限定就用Object替换。例如，类Pair<T>中的类型变量没有显式的限定，因此，原始类型用Object替换T。假定声明了一个不同的类型。

```
public class Interval<T extends Comparable & Serializable> implements Serializable
{
   public Interval(T first, T second)
   {
      if (first.compareTo(second) <= 0) { lower = first; upper = second; }
      else { lower = second; upper = first; }
   }
   . . .
   private T lower;
   private T upper;
}
```

原始类型Interval如下所示：

```
public class Interval implements Serializable
{
   public Interval(Comparable first, Comparable second) { . . . }
   . . .
```

```
    private Comparable lower;
    private Comparable upper;
  }
```

注释：读者可能想要知道切换限定：class Interval<Serializable & Comparable>会发生什么。如果这样做，原始类型用Serializable替换T，而编译器在必要时要向Comparable插入强制类型转换。为了提高效率，应该将标签（tagging）接口（即没有方法的接口）放在边界列表的末尾。

12.5.1 翻译泛型表达式

当程序调用泛型方法时，如果擦除返回类型，编译器插入强制类型转换。例如，下面这个语句序列

```
Pair<Employee> buddies = . . .;
Employee buddy = buddies.getFirst();
```

擦除getFirst的返回类型后将返回Object类型。编译器自动插入Employee的强制类型转换。也就是说，编译器把这个方法调用翻译为两条虚拟机指令：

- 对原始方法Pair.getFirst的调用。
- 将返回的Object类型强制转换为Employee类型。

当存取一个泛型域时也要插入强制类型转换。假设Pair类的first域和second域都是公有的（也许这不是一种好的编程风格，但在Java中是合法的）。表达式：

```
Employee buddy = buddies.first;
```

也会在结果字节码中插入强制类型转换。

12.5.2 翻译泛型方法

类型擦除也会出现在泛型方法中。程序员通常认为下述的泛型方法

```
public static <T extends Comparable> T min(T[] a)
```

是一个完整的方法族，而擦除类型之后，只剩下一个方法：

```
public static Comparable min(Comparable[] a)
```

注意，类型参数T已经被擦除了，只留下了限定类型 Comparable。

方法的擦除带来了两个复杂问题。看一看下面这个示例：

```
class DateInterval extends Pair<Date>
{
   public void setSecond(Date second)
   {
      if (second.compareTo(getFirst()) >= 0)
         super.setSecond(second);
   }
   . . .
}
```

一个日期区间是一对Date对象，并且需要覆盖这个方法来确保第二个值永远不小于第一个值。这个类擦除后变成

```
class DateInterval extends Pair // after erasure
{
   public void setSecond(Date second) { . . . }
   . . .
}
```

令人感到奇怪的是，存在另一个从Pair继承的setSecond方法，即

```
public void setSecond(Object second)
```

这显然是一个不同的方法，因为它有一个不同类型的参数——Object，而不是Date。然而，不应该不一样。考虑下面的语句序列：

```
DateInterval interval = new DateInterval(. . .);
Pair<Date> pair = interval; // OK--assignment to superclass
pair.setSecond(aDate);
```

这里，希望对setSecond的调用具有多态性，并调用最合适的那个方法。由于pair引用DateInterval对象，所以应该调用DateInterval.setSecond。问题在于类型擦除与多态发生了冲突。要解决这个问题，就需要编译器在DateInterval类中生成一个桥方法（bridge method）：

```
public void setSecond(Object second) { setSecond((Date) second); }
```

要想了解它的工作过程，请仔细地跟踪下列语句的执行：

```
pair.setSecond(aDate)
```

变量pair已经声明为类型Pair<Date>，并且这个类型只有一个简单的方法叫setSecond，即setSecond(Object)。虚拟机用pair引用的对象调用这个方法。这个对象是DateInterval类型的，因而将会调用DateInterval.setSecond(Object)方法。这个方法是合成的桥方法。它调用DateInterval.setSecond(Date)，这正是我们所期望的操作效果。

桥方法可能会变得十分奇怪。假设DateInterval方法也覆盖了getSecond方法：

```
class DateInterval extends Pair<Date>
{
   public Date getSecond() { return (Date) super.getSecond().clone(); }
   . . .
}
```

在擦除的类型中，有两个getSecond方法：

```
Date getSecond() // defined in DateInterval
Object getSecond() // defined in Pair
```

不能这样编写Java代码（在这里，具有相同参数类型的两个方法是不合法的）。它们都没有参数。但是，在虚拟机中，用参数类型和返回类型确定一个方法。因此，编译器可能产生两个仅返回类型不同的方法字节码，虚拟机能够正确地处理这一情况。

注释：桥方法不仅用于泛型类型。第5章已经讲过，从Java SE 5.0开始，在一个方法覆盖另一个方法时可以指定一个更严格的返回类型。例如，

```
public class Employee implements Cloneable
{
   public Employee clone() throws CloneNotSupportedException { ... }
}
```

Object.clone 和 Employee.clone方法被说成具有协变的返回类型（covariant return types）。实际上，Employee类有两个克隆方法：

```
Employee clone() // defined above
Object clone() // synthesized bridge method, overrides Object.clone
```

合成的桥方法调用了新定义的方法。

总之，需要记住有关Java泛型转换的事实：

- 虚拟机中没有泛型，只有普通的类和方法。
- 所有的类型参数都用它们的限定类型替换。
- 桥方法被合成来保持多态。
- 为保持类型安全性，必要时插入强制类型转换。

12.5.3 调用遗留代码

许多Java代码编写于Java SE 5.0之前。如果泛型类不能与这些代码互操作，就得不到广泛的应用。幸运的是，可以直接将泛型类与遗留的API中原始的对应概念放在一起使用。

下面看一个具体的示例。要想设置一个JSlider标签，可以使用方法：

```
void setLabelTable(Dictionary table)
```

在第9章中，使用下述代码填充标签表：

```
Dictionary<Integer, Component> labelTable = new Hashtable<Integer, Component>();
labelTable.put(0, new JLabel(new ImageIcon("nine.gif")));
labelTable.put(20, new JLabel(new ImageIcon("ten.gif")));
. . .
slider.setLabelTable(labelTable); // WARNING
```

在Java SE 5.0中，Dictionary和Hashtabl 类被转换成泛型类。因此，可以用Dictionary<Integer, Component> 取代原始的Dictionary。但是，当将Dictionary<Integer, Component> 对象传递给setLabelTable时，编译器会发出一个警告。

```
Dictionary<Integer, Component> labelTable = . . .;
slider.setLabelTable(labelTable); // WARNING
```

毕竟，编译器无法确定setLabelTable可能会对Dictionary对象做什么操作。这个方法可能会用字符串替换所有的关键字。这就打破了关键字类型为整型（Integer）的承诺，未来的操作有可能会产生强制类型转换的异常。

这个警告对操作不会产生什么影响，最多考虑一下JSlider有可能用Dictionary对象做什么就可以了。在这里十分清楚，JSlider只阅读这个信息，因此可以忽略这个警告。

现在，看一个相反的情形，由一个遗留的类得到一个原始类型的对象。可以将它赋给一个参数化的类型变量，当然，这样做会看到一个警告。例如，

```
Dictionary<Integer, Components> labelTable = slider.getLabelTable(); // WARNING
```

这就行了。再看一看警告，确保标签表已经包含了Integer和Component对象。当然，从来也不会有绝对的承诺。恶意的编码者可能会在滑块中设置不同的Dictionary。然而，这种情况并不会比Java SE 5.0之前的情况更糟糕。最差的情况就是程序抛出一个异常。

在查看了警告之后，可以利用注释（annotation）使之消失。注释必须放在生成这个警告的代码所在的方法之前，如下：

```
@SuppressWarnings("unchecked")
public void configureSlider() { . . . }
```

遗憾的是，这个注释也关闭了对方法内部代码的检查。将潜在不安全的代码分隔在一些不同的方法中是一个不错主意，这样可以更加容易地进行查看。

✔ **注释**：Hashtable类是抽象类Dictionary的具体子类。自从Dictionary和Hashtable类被Java SE 1.2的Map接口和HashMap类取代以来，它们都被标记为“废弃”。尽管从表面上看，它们仍然是有效的。毕竟，JSlider类是在Java SE 1.3中增加的。那时程序员不知道Map类吗？这会给在不远的未来程序员接受泛型带来希望吗？这就是与遗留代码保持一致的方式。

12.6 约束与局限性

在下面几节中，将阐述使用Java泛型时需要考虑的一些限制。大多数限制都是由类型擦除引起的。

12.6.1 不能用基本类型实例化类型参数

不能用类型参数代替基本类型。因此，没有Pair<double>，只有Pair<Double>。当然，其原因是类型擦除。擦除之后，Pair类含有Object类型的域，而Object不能存储double值。

这的确令人烦恼。但是，这样做与Java语言中基本类型的独立状态相一致。这并不是一个致命的缺陷——只有8种基本类型，当包装器类型（wrapper type）不能接受替换时，可以使用独立的类和方法处理它们。

12.6.2 运行时类型查询只适用于原始类型

虚拟机中的对象总有一个特定的非泛型类型。因此，所有的类型查询只产生原始类型。例如，

```
if (a instanceof Pair<String>) // same as a instanceof Pair
```

实际上仅仅测试a是否是任意类型的一个Pair。下面的测试同样为真

```
if (a instanceof Pair<T>) // T is ignored
```

或强制类型转换

```
Pair<String> p = (Pair<String>) a; // WARNING--can only test that a is a Pair
```

要记住这一风险，无论何时使用instanceof或涉及泛型类型的强制类型转换表达式都会看到一个编译器警告。

同样的道理，getClass方法总是返回原始类型。例如，

```
Pair<String> stringPair = . . .;
Pair<Employee> employeePair = . . .;
if (stringPair.getClass() == employeePair.getClass()) // they are equal
```

其比较的结果是true，这是因为两次调用getClass都将返回Pair.class。

12.6.3 不能抛出也不能捕获泛型类实例

不能抛出也不能捕获泛型类的对象。事实上，泛型类扩展Throwable都不合法。例如，下面的定义将不会通过编译：

```
public class Problem<T> extends Exception { /* . . . */ } // ERROR--can't extend Throwable
```

不能在catch子句中使用类型变量。例如，下面的方法将不能通过编译：

```
public static <T extends Throwable> void doWork(Class<T> t)
{
   try
   {
      do work
   }
   catch (T e) // ERROR--can't catch type variable
   {
      Logger.global.info(...)
   }
}
```

但是，在异常声明中可以使用类型变量。下面这个方法是合法的：

```
public static <T extends Throwable> void doWork(T t) throws T // OK
{
   try
   {
      do work
   }
   catch (Throwable realCause)
   {
      t.initCause(realCause);
      throw t;
   }
}
```

12.6.4 参数化类型的数组不合法

不能声明参数化类型的数组，如：

```
Pair<String>[] table = new Pair<String>[10]; // ERROR
```

这样做有什么问题呢？擦除之后，table的类型是Pair[]，可以将其转换为Object[]：

```
Object[] objarray = table;
```

数组能够记住它的元素类型（component type），如果试图存入一个错误类型的元素，就会抛出一个ArrayStoreException异常：

```
objarray[0] = "Hello"; // ERROR--component type is Pair
```

但是，对于泛型而言，擦除将会降低这一机制的效率。赋值

```
objarray[0] = new Pair<Employee>();
```

可以通过数组存储的检测，但仍然会导致类型错误。因此，禁止使用参数化类型的数组。

提示： 如果需要收集参数化类型的对象，最好直接使用ArrayList: ArrayList <Pair<String>>，这样既安全又有效。

12.6.5 不能实例化类型变量

不能使用像new T(...)，new T[...]或T.class这样的表达式中的类型变量。例如，下面的Pair<T>构造器就是非法的：

```
public Pair() { first = new T(); second = new T(); } // ERROR
```

类型擦除将T改变成Object，而且，本意肯定不希望调用new Object( )。但是，可以通过反射调用Class.newInstance方法来构造泛型对象。遗憾的是，细节有点复杂。不能调用：

```
first = T.class.newInstance(); // ERROR
```

表达式T.class是不合法的。必须像下面这样设计API以便可以支配Class 对象：

```
public static <T> Pair<T> makePair(Class<T> cl)
{
   try { return new Pair<T>(cl.newInstance(), cl.newInstance()) }
   catch (Exception ex) { return null; }
}
```

这个方法可以按照下列方式调用：

```
Pair<String> p = Pair.makePair(String.class);
```

注意，Class类本身是泛型。例如，String.class是一个Class<String>的实例（事实上，它是惟一的实例）。因此，makePair方法能够推断出pair的类型。

不能构造一个泛型数组：

```
public static <T extends Comparable> T[] minmax(T[] a) { T[] mm = new T[2]; . . . } // ERROR
```

类型擦除会让这个方法永远构造Object[2]数组。

如果数组仅仅作为一个类的私有实例域，就可以将这个数组声明为Object[]，并且在获取元素时进行类型转换。例如，ArrayList类可以这样实现：

```
public class ArrayList<E>
{
   private Object[] elements;
   @SuppressWarnings("unchecked") public E get(int n) { return (E) elements[n]; }
   public void set(int n, E e) { elements[n] = e; } // no cast needed
   . . .
}
```

实际的实现没有这么清晰：

```
public class ArrayList<E>
{
   private E[] elements;
   public ArrayList() { elements = (E[]) new Object[10]; }
   . . .
}
```

这里，强制类型转换E[ ]是一个假象，而类型擦除使其无法察觉。

由于minmax方法返回T[ ]数组，使得这一技术无法施展，如果掩盖这个类型会有运行时错误结果。假设执行代码：

```
public static <T extends Comparable> T[] minmax(T[] a)
{
   Object[] mm = new Object[2];
   . . .;
   return (T[]) mm; // compiles with warning
}
```

调用

```
String[] ss = minmax("Tom", "Dick", "Harry");
```

编译时不会有任何警告。当Object[]引用赋给String[]变量时，将会发生ClassCastException异常。

在这种情况下，可以利用反射，调用Array.newInstance:

```
public static <T extends Comparable> T[] minmax(T[] a)
{
   T[] mm = (T[]) Array.newInstance(a.getClass().getComponentType(), 2);
   . . .
}
```

ArrayList类的toArray方法就没有这么幸运。它需要生成一个T[]数组，但没有成分类型。因此，有下面两种不同的形式：

```
Object[] toArray()
T[] toArray(T[] result)
```

第二个方法接收一个数组参数。如果数组足够大，就使用这个数组。否则，用result的成分类型构造一个足够大的新数组。

12.6.6 泛型类的静态上下文中类型变量无效

不能在静态域或方法中引用类型变量。例如，下列高招将无法施展：

```
public class Singleton<T>
{
   public static T getSingleInstance() // ERROR
   {
      if (singleInstance == null) construct new instance of T
      return singleInstance;
   }
   private static T singleInstance; // ERROR
}
```

如果这个程序能够运行，就可以声明一个Singleton<Random>共享随机数生成器，声明一个Singleton<JFileChooser>共享文件选择器对话框。但是，这个程序无法工作。类型擦除之后，只剩下Singleton类，它只包含一个singleInstance域。因此，禁止使用带有类型变量的静态域和方法。

12.6.7 注意擦除后的冲突

当泛型类型被擦除时，无法创建引发冲突的条件。下面是一个示例。假定像下面这样将

equals方法添加到Pair类中：

```
public class Pair<T>
{
   public boolean equals(T value) { return first.equals(value) && second.equals(value); }
   . . .
}
```

考虑一个Pair<String>。从概念上讲，它有两个equals方法：

```
boolean equals(String) // defined in Pair<T>
boolean equals(Object) // inherited from Object
```

但是，直觉把我们引入歧途。方法擦除

```
boolean equals(T)
```

就是

```
boolean equals(Object)
```

与Object.equals方法发生冲突。

当然，补救的办法是重新命名引发错误的方法。

泛型规范说明还提到另外一个原则："要想支持擦除的转换，就需要强行限制一个类或类型变量不能同时成为两个接口类型的子类，而这两个接口是同一接口的不同参数化。"例如，下述代码是非法的：

```
class Calendar implements Comparable<Calendar> { . . . }
class GregorianCalendar extends Calendar implements Comparable<GregorianCalendar>
  { . . . } // ERROR
```

GregorianCalendar会实现Comparable<Calendar>和Comparable<GregorianCalendar >，这是同一接口的不同参数化。

这一限制与类型擦除的关系并不十分明确。毕竟，下列非泛型版本是合法的。

```
class Calendar implements Comparable { . . . }
class GregorianCalendar extends Calendar implements Comparable { . . . }
```

其原因非常微妙，有可能与合成的桥方法产生冲突。实现了Comparable<X>的类可以获得一个桥方法：

```
public int compareTo(Object other) { return compareTo((X) other); }
```

对于不同类型的X不能有两个这样的方法。

12.7 泛型类型的继承规则

在使用泛型类时，需要了解一些有关继承和子类型的准则。下面先从许多程序员感觉不太直观的情况开始。考虑一个类和一个子类，如Employee和Manager。Pair<Manager>是Pair<Employee>的一个子类吗？答案是"不是"，或许人们会感到奇怪。例如，下面的代码将不能编译成功：

```
Manager[] topHonchos = . . .;
Pair<Employee> result = ArrayAlg.minmax(topHonchos); // ERROR
```

minmax方法返回Pair<Manager>，而不是Pair<Employee>，并且这样的赋值是不合法的。无论S与T有什么联系（如图12-1所示），通常，Pair<S>与Pair<T>没有什么联系。

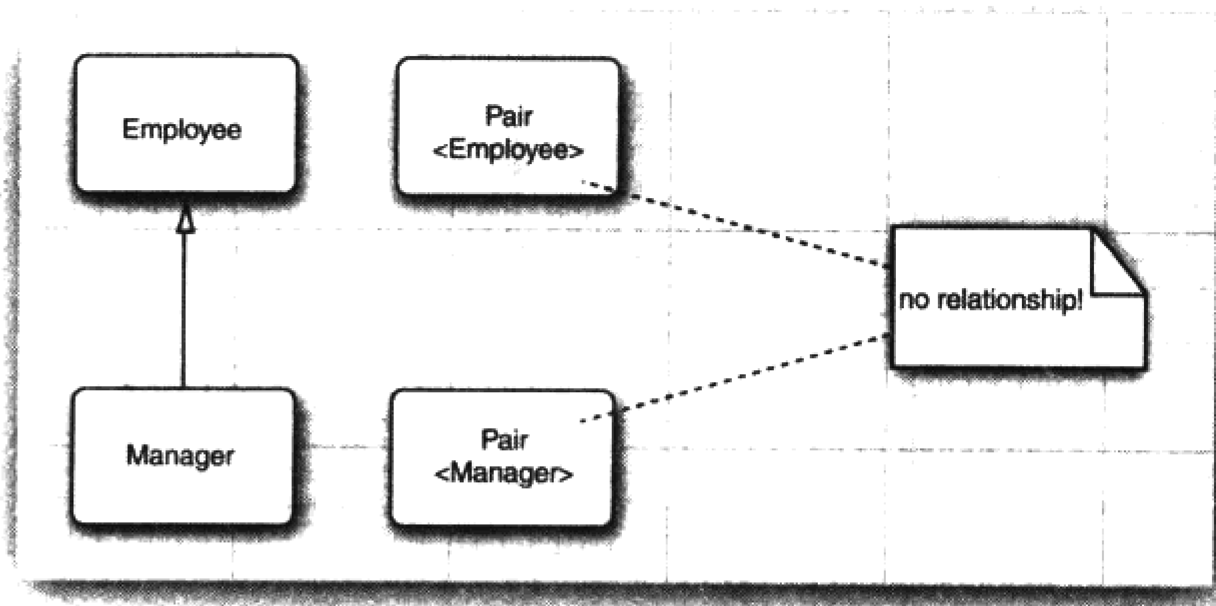


图12-1 pair类之间没有继承关系

这一限制看起来过于严格，但对于类型安全非常必要。假设允许将Pair<Manager>转换为Pair<Employee>。考虑下面代码：

```
Pair<Manager> managerBuddies = new Pair<Manager>(ceo, cfo);
Pair<Employee> employeeBuddies = managerBuddies; // illegal, but suppose it wasn't
employeeBuddies.setFirst(lowlyEmployee);
```

显然，最后一句是合法的。但是employeeBuddies和managerBuddies引用了同样的对象。现在将CFO和一个普通员工组成一对，这对于Pair<Manager>来说应该是不可能的。

> **注释**：*必须注意泛型与Java数组之间的重要区别。可以将一个Manager[]数组赋给一个类型为Employee[]的变量：*
>
> ```
> Manager[] managerBuddies = { ceo, cfo };
> Employee[] employeeBuddies = managerBuddies; // OK
> ```
>
> *然而，数组带有特别的保护。如果试图将一个低级别的雇员存储到employeeBuddies[0]，虚拟机将会抛出ArrayStoreException异常。*

永远可以将参数化类型转换为一个原始类型。例如，Pair<Employee>是原始类型Pair的一个子类型。在与遗留代码衔接时，这个转换非常必要。

转换成原始类型之后会产生类型错误吗？很遗憾，会！看一看下面这个示例：

```
Pair<Manager> managerBuddies = new Pair<Manager>(ceo, cfo);
Pair rawBuddies = managerBuddies; // OK
rawBuddies.setFirst(new File(". . .")); // only a compile-time warning
```

听起来有点吓人。但是，请记住现在的状况不会再比旧版Java的情况糟糕。虚拟机的安全性还没有到生死攸关的程度。当使用getFirst获得外来对象并赋给Manager变量时，与通常一样，会抛出ClassCastException异常。这里失去的只是泛型程序设计提供的附加安全性。

最后，泛型类可以扩展或实现其他的泛型类。就这一点而言，与普通的类没有什么区别。

例如，ArrayList<T>类实现List<T>接口。这意味着，一个ArrayList<Manager>可以被转换为一个List<Manager>。但是，如前面所见，一个ArrayList<Manager>不是一个ArrayList <Employee>或List<Employee>。图12-2展示了它们之间的联系。

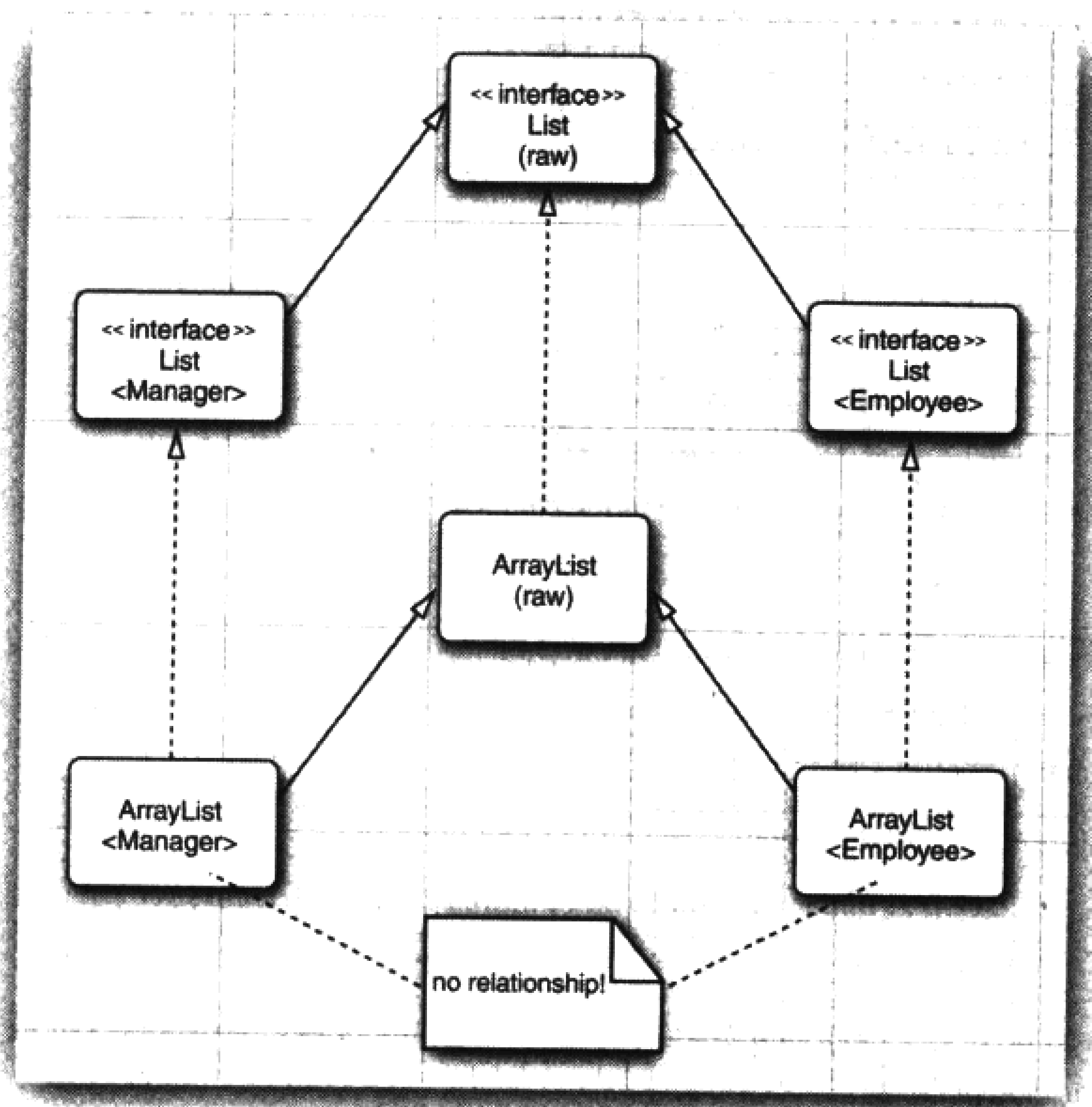


图12-2 泛型列表类型中子类型间的联系

12.8 通配符类型

固定的泛型类型系统使用起来并没有那么令人愉快，类型系统的研究人员知道这一点已经有一段时间了。Java的设计者发明了一种巧妙的（仍然是安全的）"解决方案"：通配符类型。例如，通配符类型

```
Pair<? extends Employee>
```

表示任何泛型Pair类型，它的类型参数是Employee的子类，如Pair<Manager>，但不是Pair<String>。

假设要编写一个打印雇员对的方法，像这样：

```
public static void printBuddies(Pair<Employee> p)
{
   Employee first = p.getFirst();
   Employee second = p.getSecond();
```

```
    System.out.println(first.getName() + " and " + second.getName() + " are buddies.";
}
```

正如前面讲到的，不能将Pair<Manager>传递给这个方法，这一点很受限制。解决的方法很简单：使用通配符类型：

```
public static void printBuddies(Pair<? extends Employee> p)
```

类型Pair<Manager>是Pair<? extends Employee>的子类型（参看图12-3）。

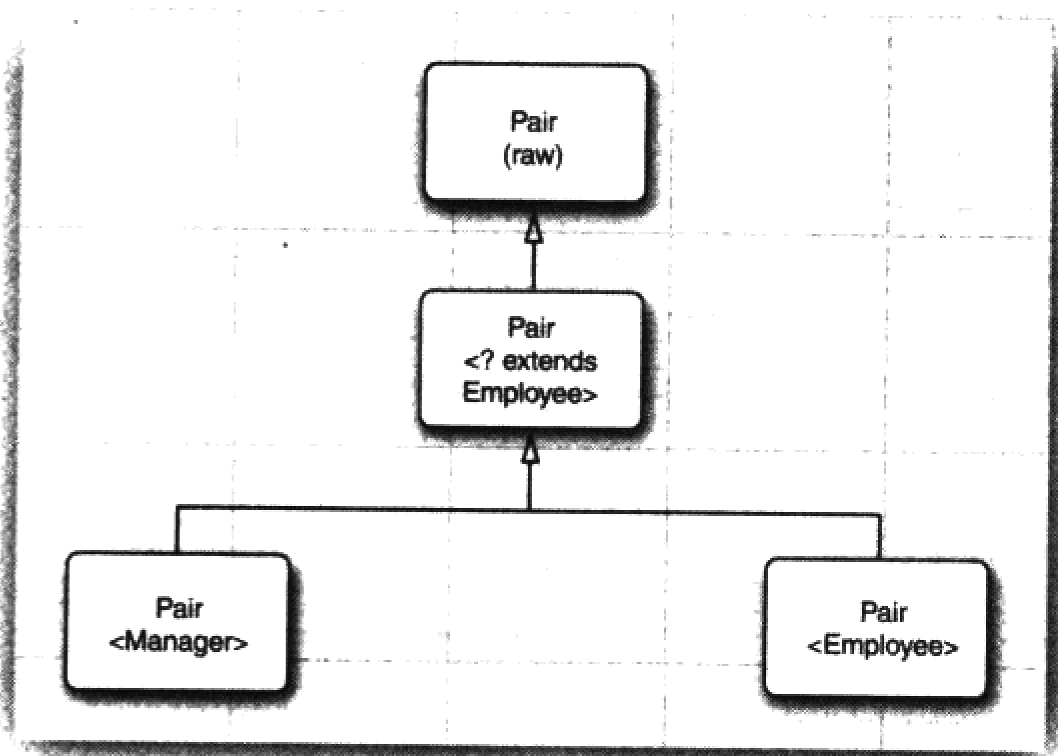


图12-3 使用通配符的子类型的联系

使用通配符会通过Pair<? extends Employee>的引用破坏Pair<Manager>吗？

```
Pair<Manager> managerBuddies = new Pair<Manager>(ceo, cfo);
Pair<? extends Employee> wildcardBuddies = managerBuddies; // OK
wildcardBuddies.setFirst(lowlyEmployee); // compile-time error
```

这可能不会引起破坏。对setFirst的调用有一个类型错误。要了解其中的缘由，请仔细看一看类型Pair<? extends Employee>。其方法似乎是这样的：

```
? extends Employee getFirst()
void setFirst(? extends Employee)
```

这样将不可能调用setFirst方法。编译器只知道需要某个Employee的子类型，但不知道具体是什么类型。它拒绝传递任何特定的类型。毕竟，?不能用来匹配。

使用getFirst就不存在这个问题：将getFirst的返回值赋给一个Employee的引用完全合法。

这就是引入有限定的通配符的关键之处。现在已经有办法区分安全的访问器方法和不安全的更改器方法了。

12.8.1 通配符的超类型限定

通配符限定与类型变量限定十分类似，但是，还有一个附加的能力，即可以指定一个超类型限定（supertype bound），如下所示：

```
? super Manager
```

这个通配符限制为Manager的所有超类型。(已有的super关键字十分准确地描述了这种联系，这一点十分令人感到欣慰。)

为什么要这样做呢？带有超类型限定的通配符的行为与前面介绍的相反。可以为方法提供参数，但不能使用返回值。例如，Pair<? super Manager>有方法

```
void setFirst(? super Manager)
? super Manager getFirst()
```

编译器不知道setFirst方法的确切类型，但是可以用任意Manager对象（或子类型，例如，Executive）调用它，而不能用Employee对象调用。然而，如果调用getFirst，返回的对象类型就不会得到保证。只能把它赋给一个Object。

下面是一个典型的示例。有一个经理的数组，并且想把奖金最高和最低的经理放在一个Pair对象中。Pair的类型是什么？在这里，Pair<Employee>是合理的，Pair<Object>也是合理的（如图12-4所示）。下面的方法将可以接受任何适当的Pair：

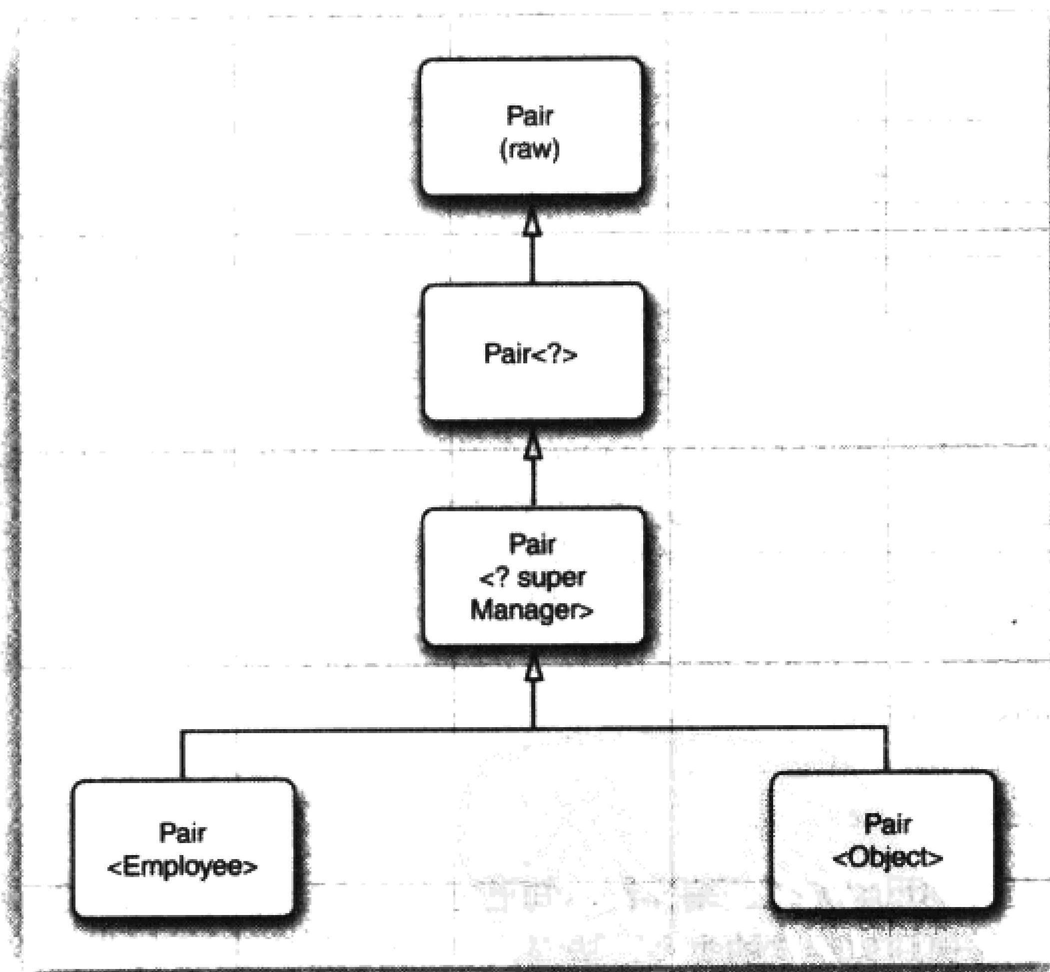


图12-4 带有超类型边界的通配符

```
public static void minmaxBonus(Manager[] a, Pair<? super Manager> result)
{
   if (a == null || a.length == 0) return;
   Manager min = a[0];
   Manager max = a[0];
   for (int i = 1; i < a.length; i++)
   {
```

```
        if (min.getBonus() > a[i].getBonus()) min = a[i];
        if (max.getBonus() < a[i].getBonus()) max = a[i];
    }
    result.setFirst(min);
    result.setSecond(max);
}
```

直观地讲，带有超类型限定的通配符可以向泛型对象写入，带有子类型限定的通配符可以从泛型对象读取。

下面是超类型限定的另一种应用。Comparable接口本身就是一个泛型类型。声明如下：

```
public interface Comparable<T>
{
    public int compareTo(T other);
}
```

在此，类型变量指示了other参数的类型。例如，String类实现Comparable <String>，它的compareTo方法被声明为

```
public int compareTo(String other)
```

很好，显式的参数有一个正确的类型。在Java SE 5.0之前，other是一个Object，并且这个方法的实现需要强制类型转换。

由于Comparable是一个泛型类型，也许可以把ArrayAlg类的min方法做得更好一些？可以这样声明：

```
public static <T extends Comparable<T>> T min(T[] a)
```

看起来，这样写比只使用T extents Comparable更彻底，并且对许多类来讲，工作得更好。例如，如果计算一个String数组的最小值，T就是String类型的，而String是Comparable<String>的子类型。但是，当处理一个GregorianCalendar对象的数组时，就会出现问题。GregorianCalendar是Calendar的子类，并且Calendar实现了Comparable<Calendar>。因此GregorianCalendar实现的是Comparable<Calendar>，而不是Comparable<GregorianCalendar>。

在这种情况下，超类型可以用来进行救助：

```
public static <T extends Comparable<? super T>> T min(T[] a) . . .
```

现在compareTo方法写成

```
int compareTo(? super T)
```

有可能被声明为使用类型T的对象，也有可能使用T的超类型（例如，当T是GregorianCalendar）。无论如何，传递一个T类型的对象给compareTo方法都是安全的。

对于初学者来说，<T extends Comparable<? super T>>这样的声明看起来有点吓人。很遗憾，因为这一声明的意图在于帮助应用程序员排除调用参数上的不必要的限制。对泛型没有兴趣的应用程序员很可能很快就学会掩盖这些声明，想当然地认为库程序员做的都是正确的。如果是一名库程序员，一定要习惯于通配符，否则，就会受到用户的责备，还要在代码中随意地添加强制类型转换直至代码可以编译。

12.8.2 无限定通配符

还可以使用无限定的通配符，例如，Pair<?>。初看起来，这好像与原始的Pair类型一样。实际上，有很大的不同。类型Pair<?>有方法如：

```
? getFirst()
void setFirst(?)
```

getFirst的返回值只能赋给一个Object。setFirst方法不能被调用，甚至不能用Object调用。Pair<?>和Pair本质的不同在于：可以用任意Object对象调用原始的Pair类的setObject方法。

为什么要使用这样脆弱的类型？它对于许多简单的操作非常有用。例如，下面这个方法将用来测试一个pair是否包含了指定的对象，它不需要实际的类型。

```
public static boolean hasNulls(Pair<?> p)
{
   return p.getFirst() == null || p.getSecond() == null;
}
```

通过将contains转换成泛型方法，可以避免使用通配符类型：

```
public static <T> boolean hasNulls(Pair<T> p)
```

但是，带有通配符的版本可读性更强。

12.8.3 通配符捕获

编写一个交换一个pair元素的方法：

```
public static void swap(Pair<?> p)
```

通配符不是类型变量，因此，不能在编写代码中使用“?”作为一种类型。也就是说，下述代码是非法的：

```
? t = p.getFirst(); // ERROR
p.setFirst(p.getSecond());
p.setSecond(t);
```

这是一个问题，因为在交换的时候必须临时保存第一个元素。幸运的是，这个问题有一个有趣的解决方案。我们可以写一个辅助方法swapHelper，如下所示：

```
public static <T> void swapHelper(Pair<T> p)
{
   T t = p.getFirst();
   p.setFirst(p.getSecond());
   p.setSecond(t);
}
```

注意，swapHelper是一个泛型方法，而swap不是，它具有固定的Pair<?>类型的参数。

现在可以由swap调用swapHelper：

```
public static void swap(Pair<?> p) { swapHelper(p); }
```

在这种情况下，swapHelper方法的参数T捕获通配符。它不知道是哪种类型的通配符，但是，这是一个明确的类型，并且<T>swapHelper的定义只有在T指出类型时才有明确的含义。

当然，在这种情况下，并不是一定要使用通配符。我们已经直接实现了没有通配符的泛型

方法<T> void swap(Pair<T> p)。然而，下面看一个通配符类型出现在计算中间的示例：

```
public static void maxminBonus(Manager[] a, Pair<? super Manager> result)
{
   minmaxBonus(a, result);
   PairAlg.swapHelper(result); // OK--swapHelper captures wildcard type
}
```

在这里，通配符捕获机制是不可避免的。

通配符捕获只有在有许多限制的情况下才是合法的。编译器必须能够确信通配符表达的是单个、确定的类型。例如，ArrayList<Pair<T>>中的T永远不能捕获ArrayList<Pair<?>>中的通配符。数组列表可以保存两个Pair<?>，分别针对?的不同类型。

例12-3中的测试程序将前几节讨论的各种方法综合在一起，读者从中可以看到它们彼此之间的关联。

例12-3 PairTest3.java

```
import java.util.*;

/**
 * @version 1.00 2004-05-10
 * @author Cay Horstmann
 */
public class PairTest3
{
   public static void main(String[] args)
   {
      Manager ceo = new Manager("Gus Greedy", 800000, 2003, 12, 15);
      Manager cfo = new Manager("Sid Sneaky", 600000, 2003, 12, 15);
      Pair<Manager> buddies = new Pair<Manager>(ceo, cfo);
      printBuddies(buddies);

      ceo.setBonus(1000000);
      cfo.setBonus(500000);
      Manager[] managers = { ceo, cfo };

      Pair<Employee> result = new Pair<Employee>();
      minmaxBonus(managers, result);
      System.out.println("first: " + result.getFirst().getName()
         + ", second: " + result.getSecond().getName());
      maxminBonus(managers, result);
      System.out.println("first: " + result.getFirst().getName()
         + ", second: " + result.getSecond().getName());
   }

   public static void printBuddies(Pair<? extends Employee> p)
   {
      Employee first = p.getFirst();
      Employee second = p.getSecond();
      System.out.println(first.getName() + " and " + second.getName() + " are buddies.");
   }

   public static void minmaxBonus(Manager[] a, Pair<? super Manager> result)
   {
```

```
      if (a == null || a.length == 0) return;
      Manager min = a[0];
      Manager max = a[0];
      for (int i = 1; i < a.length; i++)
      {
         if (min.getBonus() > a[i].getBonus()) min = a[i];
         if (max.getBonus() < a[i].getBonus()) max = a[i];
      }
      result.setFirst(min);
      result.setSecond(max);
   }

   public static void maxminBonus(Manager[] a, Pair<? super Manager> result)
   {
      minmaxBonus(a, result);
      PairAlg.swapHelper(result); // OK--swapHelper captures wildcard type
   }
}

class PairAlg
{
   public static boolean hasNulls(Pair<?> p)
   {
      return p.getFirst() == null || p.getSecond() == null;
   }

   public static void swap(Pair<?> p) { swapHelper(p); }

   public static <T> void swapHelper(Pair<T> p)
   {
      T t = p.getFirst();
      p.setFirst(p.getSecond());
      p.setSecond(t);
   }
}

class Employee
{
   public Employee(String n, double s, int year, int month, int day)
   {
      name = n;
      salary = s;
      GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
      hireDay = calendar.getTime();
   }

   public String getName()
   {
      return name;
   }

   public double getSalary()
   {
      return salary;
   }

   public Date getHireDay()
```

```
   {
      return hireDay;
   }

   public void raiseSalary(double byPercent)
   {
      double raise = salary * byPercent / 100;
      salary += raise;
   }

   private String name;
   private double salary;
   private Date hireDay;
}

class Manager extends Employee
{
   /**
      @param n the employee's name
      @param s the salary
      @param year the hire year
      @param month the hire month
      @param day the hire day
   */
   public Manager(String n, double s, int year, int month, int day)
   {
      super(n, s, year, month, day);
      bonus = 0;
   }

   public double getSalary()
   {
      double baseSalary = super.getSalary();
      return baseSalary + bonus;
   }

   public void setBonus(double b)
   {
      bonus = b;
   }

   public double getBonus()
   {
      return bonus;
   }

   private double bonus;
}
```

12.9 反射和泛型

现在，Class类是泛型的。例如，String.class实际上是一个Class<String>类的对象（事实上，是惟一的对象）。

类型参数十分有用，这是因为它允许Class<T>方法的返回类型更加具有针对性。下面

Class<T>中的方法就使用了类型参数：

```
T newInstance()
T cast(Object obj)
T[] getEnumConstants()
Class<? super T> getSuperclass()
Constructor<T> getConstructor(Class... parameterTypes)
Constructor<T> getDeclaredConstructor(Class... parameterTypes)
```

newInstance方法返回一个实例，这个实例所属的类由默认的构造器获得。它的返回类型目前被声明为T，其类型与Class<T>描述的类相同，这样就免除了类型转换。

如果给定的类型确实是T的一个子类型，cast方法就会返回一个现在声明为类型T的对象，否则，抛出一个BadCastException异常。

如果这个类不是enum类或类型T的枚举值的数组，getEnumConstants方法将返回null。

最后，getConstructor与getdeclaredConstructor方法返回一个Constructor<T>对象。Constructor类也已经变成泛型，以便newInstance方法有一个正确的返回类型。

API java.lang.Class<T> 1.0

- `T newInstance() 5.0`

  返回默认构造器构造的一个新实例。
- `T cast(Object obj) 5.0`

  如果obj为null或有可能转换成类型T，则返回obj；否则抛出BadCastException异常。
- `T[ ] getEnumConstants( ) 5.0`

  如果T是枚举类型，则返回所有值组成的数组，否则返回null。
- `Class<? super T> getSuperclass( ) 5.0`

  返回这个类的超类。如果T不是一个类或Object类，则返回null。
- `Constructor<T> getConstructor(Class. . . parameterTypes) 5.0`
- `Constructor<T> getDeclaredConstructor(Class. . . parameterTypes) 5.0`

  获得公有的构造器，或带有给定参数类型的构造器。

API java.lang.reflect.Constructor<T> 1.1

- `T newInstance(Object... parameters) 5.0`

  返回用指定参数构造的新实例。

12.9.1 使用Class<T>参数进行类型匹配

有时，匹配泛型方法中的Class<T>参数的类型变量很有实用价值。下面是一个具有一定权威的示例：

```
public static <T> Pair<T> makePair(Class<T> c) throws InstantiationException,
    IllegalAccessException
{
   return new Pair<T>(c.newInstance(), c.newInstance());
}
```

如果调用

```
makePair(Employee.class)
```

Employee.class是类型Class<Employee>的一个对象。makePair方法的类型参数T同Employee匹配，并且编译器可以推断出这个方法将返回一个Pair<Employee>。

12.9.2 虚拟机中的泛型类型信息

Java泛型的卓越特性之一是在虚拟机中泛型类型的擦除。令人感到奇怪的是，擦除的类仍然保留一些泛型祖先的微弱记忆。例如，原始的Pair类知道源于泛型类Pair<T>，即使一个Pair类型的对象无法区分是由Pair<String>构造的还是由Pair<Employee>构造的。

类似地，看一下方法

```
public static Comparable min(Comparable[] a)
```

这是一个泛型方法的擦除

```
public static <T extends Comparable<? super T>> T min(T[] a)
```

可以使用Java SE 5.0增加的反射API来确定：

- 这个泛型方法有一个叫做T的类型参数。
- 这个类型参数有一个子类型限定，其自身又是一个泛型类型。
- 这个限定类型有一个通配符参数。
- 这个通配符参数有一个超类型限定。
- 这个泛型方法有一个泛型数组参数。

换句话说，需要重新构造实现者声明的泛型类以及方法中的所有内容。但是，不会知道对于特定的对象或方法调用，如何解释类型参数。

 **注释**：包含在类文件中，让泛型反射可用的类型信息与旧的虚拟机不兼容。

为了表达泛型类型声明，Java SE 5.0在java.lang.reflect包中提供了一个新的接口Type。这个接口包含下列子类型：

- Class类，描述具体类型。
- TypeVariable接口，描述类型变量（如T extends Comparable<? super T>）。
- WildcardType接口，描述通配符（如? super T）。
- ParameterizedType接口，描述泛型类或接口类型（如Comparable<? super T>）。
- GenericArrayType接口，描述泛型数组（如T[ ]）。

图12-5给出了继承层次。注意，最后4个子类型是接口，虚拟机将实例化实现这些接口的适当的类。

例12-4使用泛型反射API打印出给定类的有关内容。如果用Pair类运行，将会得到下列报告：

```
class Pair<T> extends java.lang.Object
public T getFirst()
public T getSecond()
public void setFirst(T)
public void setSecond(T)
```

如果使用PairTest2目录下的ArrayAlg运行，将会得到下列报告：

```
public static <T extends java.lang.Comparable> Pair<T> minmax(T[])
```

本节末尾的API注释描述了示例程序中使用的这些方法。

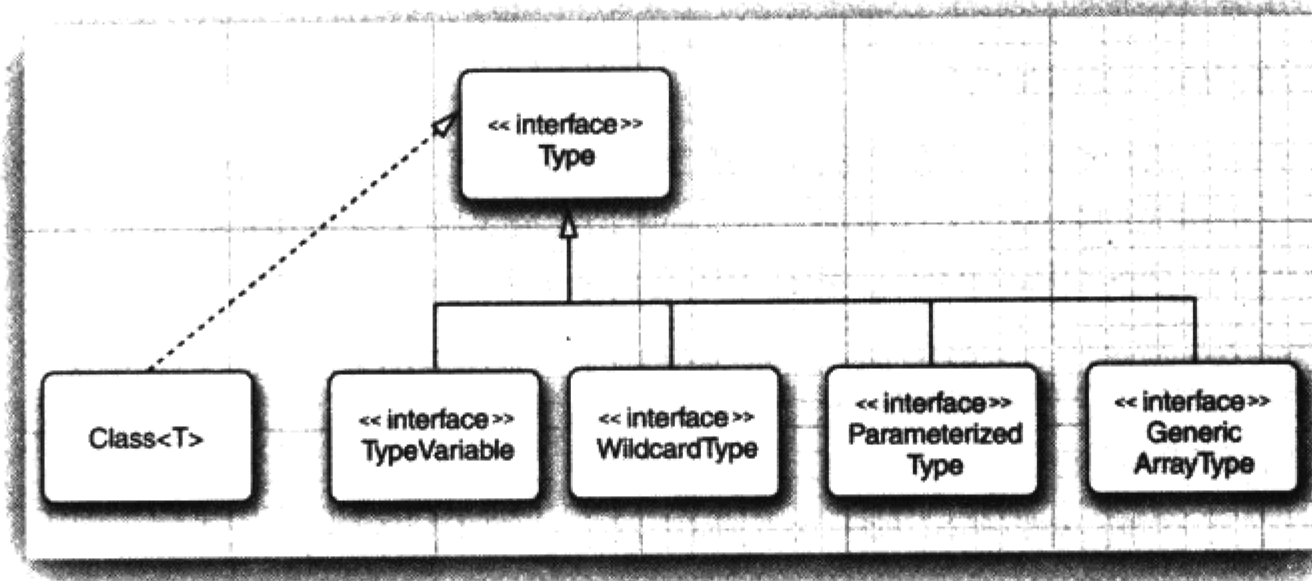


图12-5 Type类和它的后代

例12-4 GenericReflectionTest.java

```
import java.lang.reflect.*;
import java.util.*;

/**
 * @version 1.10 2004-05-15
 * @author Cay Horstmann
 */
public class GenericReflectionTest
{
   public static void main(String[] args)
   {
      // read class name from command-line args or user input
      String name;
      if (args.length > 0) name = args[0];
      else
      {
         Scanner in = new Scanner(System.in);
         System.out.println("Enter class name (e.g. java.util.Collections): ");
         name = in.next();
      }

      try
      {
         // print generic info for class and public methods
         Class cl = Class.forName(name);
         printClass(cl);
         for (Method m : cl.getDeclaredMethods())
            printMethod(m);
      }
      catch (ClassNotFoundException e)
      {
```

```
         e.printStackTrace();
      }
   }

   public static void printClass(Class cl)
   {
      System.out.print(cl);
      printTypes(cl.getTypeParameters(), "<", ", ", ">", true);
      Type sc = cl.getGenericSuperclass();
      if (sc != null)
      {
         System.out.print(" extends ");
         printType(sc, false);
      }
      printTypes(cl.getGenericInterfaces(), " implements ", ", ", "", false);
      System.out.println();
   }

   public static void printMethod(Method m)
   {
      String name = m.getName();
      System.out.print(Modifier.toString(m.getModifiers()));
      System.out.print(" ");
      printTypes(m.getTypeParameters(), "<", ", ", "> ", true);

      printType(m.getGenericReturnType(), false);
      System.out.print(" ");
      System.out.print(name);
      System.out.print("(");
      printTypes(m.getGenericParameterTypes(), "", ", ", "", false);
      System.out.println(")");
   }

   public static void printTypes(Type[] types, String pre, String sep, String suf,
         boolean isDefinition)
   {
      if (pre.equals(" extends ") && Arrays.equals(types, new Type[] { Object.class })) return;
      if (types.length > 0) System.out.print(pre);
      for (int i = 0; i < types.length; i++)
      {
         if (i > 0) System.out.print(sep);
         printType(types[i], isDefinition);
      }
      if (types.length > 0) System.out.print(suf);
   }

   public static void printType(Type type, boolean isDefinition)
   {
      if (type instanceof Class)
      {
         Class t = (Class) type;
         System.out.print(t.getName());
      }
      else if (type instanceof TypeVariable)
      {
         TypeVariable t = (TypeVariable) type;
         System.out.print(t.getName());
```

```
            if (isDefinition)
               printTypes(t.getBounds(), " extends ", " & ", "", false);
         }
         else if (type instanceof WildcardType)
         {
            WildcardType t = (WildcardType) type;
            System.out.print("?");
            printTypes(t.getUpperBounds(), " extends ", " & ", "", false);
            printTypes(t.getLowerBounds(), " super ", " & ", "", false);
         }
         else if (type instanceof ParameterizedType)
         {
            ParameterizedType t = (ParameterizedType) type;
            Type owner = t.getOwnerType();
            if (owner != null)
            {
               printType(owner, false);
               System.out.print(".");
            }
            printType(t.getRawType(), false);
            printTypes(t.getActualTypeArguments(), "<", ", ", ">", false);
         }
         else if (type instanceof GenericArrayType)
         {
            GenericArrayType t = (GenericArrayType) type;
            System.out.print("");
            printType(t.getGenericComponentType(), isDefinition);
            System.out.print("[]");
         }

      }
}
```

API java.lang.Class<T> 1.0

- `TypeVariable[ ] getTypeParameters( ) 5.0`

  如果这个类型被声明为泛型类型，则获得泛型类型变量，否则获得一个长度为0的数组。

- `Type getGenericSuperclass( ) 5.0`

  获得被声明为这一类型的超类的泛型类型；如果这个类型是Object或不是一个类类型(class type)，则返回null。

- `Type[ ] getGenericInterfaces( ) 5.0`

  获得被声明为这个类型的接口的泛型类型（以声明的次序），否则，如果这个类型没有实现接口，返回长度为0的数组。

API java.lang.reflect.Method 1.1

- `TypeVariable[ ] getTypeParameters( ) 5.0`

  如果这个方法被声明为泛型方法，则获得泛型类型变量，否则返回长度为0的数组。

- `Type getGenericReturnType( ) 5.0`

获得这个方法被声明的泛型返回类型。

• Type[ ] getGenericParameterTypes( ) 5.0

获得这个方法被声明的泛型参数类型。如果这个方法没有参数，返回长度为0的数组。

API Java.lang.reflect.TypeVariable 5.0

• String getName( )

获得类型变量的名字。

• Type[ ] getBounds( )

获得类型变量的子类限定，否则，如果该变量无限定，则返回长度为0的数组。

API Java.lang.reflect.WildcardType> 5.0

• Type[ ] getLowerBounds( )

获得这个类型变量的子类（extends）限定，否则，如果没有子类限定，则返回长度为0的数组。

• Type[ ] getUpperBounds( )

获得这个类型变量的超类（super）限定，否则，如果没有超类限定，则返回长度为0的数组。

API Java.lang.reflect.ParameterizedType 5.0

• Type getRawType( )

获得这个参数化类型的原始类型。

• Type[ ] getActualTypeArguments( )

获得这个参数化类型声明时所使用的类型参数。

• Type getOwnerType( )

如果是内部类型，则返回其外部类型，如果是一个顶级类型，则返回null。

API Java.lang.reflect.GenericArrayType 5.0

• Type getGenericComponentType( )

获得声明该数组类型的泛型组合类型。

现在已经学习了如何使用泛型类以及在必要时如何自定义泛型类和泛型方法。同样重要的是，学习了如何解译在API文档和错误消息中遇到的泛型类型声明。要想了解有关Java泛型更加详尽的信息，可以到http://angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html上求助。那里有一个很好的常见问题解答列表（也有一些不太常见的）。

在下一章中，将学习Java集合框架如何使用泛型。

第13章 集 合

▲ 集合接口
▲ 具体的集合
▲ 集合框架
▲ 算法
▲ 遗留的集合

在实现方法时，选择不同的数据结构会导致其实现风格以及性能存在着很大差异。需要快速地搜索成千上万个（甚至上百万）有序的数据项吗？需要快速地在有序的序列中间插入元素或删除元素吗？需要建立键与值之间的关联吗？

本章将讲述如何利用Java类库帮助我们在程序设计中实现传统的数据结构。在大学的计算机科学课程中，有一门叫做*数据结构*（Data Structures）的课程，通常要讲授一个学期，因此，有许许多多专门探讨这个重要主题的书籍。与大学课程所讲述的内容不同，这里，将跳过理论部分，仅介绍如何使用标准库中的集合类。

13.1 集合接口

Java最初版本只为最常用的数据结构提供了很少的一组类：Vector、Stack、Hashtable、BitSet、与Enumeration接口，其中的Enumeration接口提供了一种用于访问任意容器中各个元素的抽象机制。这是一种很明智的选择，但要想建立一个全面的集合类库还需要大量的时间和高超的技能。

随着Java SE 1.2的问世，设计人员感到是推出一组功能完善的数据结构的时机了。面对一大堆相互矛盾的设计策略，他们希望让类库规模小且易于学习，而不希望像C++的“标准模版库”（即STL）那样复杂，但却又希望能够得到STL率先推出的“泛型算法”所具有的优点。他们希望将传统的类融入新的框架中。与所有的集合类库设计者一样，他们必须做出一些艰难的选择，于是，在整个设计过程中，他们做出了一些独具特色的设计决定。本节将介绍Java集合框架的基本设计，展示使用它们的方法，并解释一些颇具争议的特性背后的考虑。

13.1.1 将集合的接口与实现分离

与现代的数据结构类库的常见情况一样，Java集合类库也将*接口*（interfaces）与*实现*（implementations）分离。首先，看一下人们熟悉的数据结构——队列（queue）是如何分离的。

*队列接口*指出可以在队列的尾部添加元素，在队列的头部删除元素，并且可以查找队列中元素的个数。当需要收集对象，并按照“先进先出”的规则检索对象时就应该使用队列（见图13-1）。

一个队列接口的最小形式可能类似下面这样：

```
interface Queue<E> // a simplified form of the interface in the standard library
{
   void add(E element);
   E remove();
   int size();
}
```

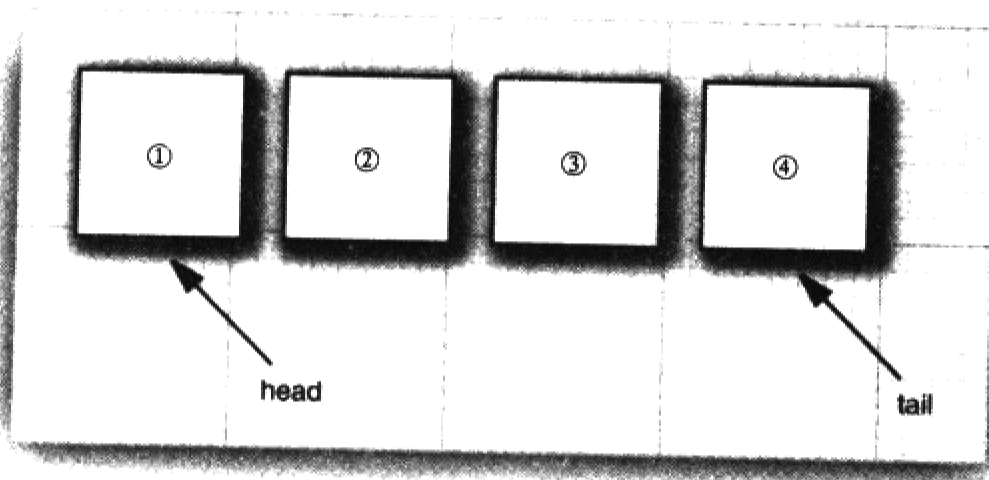


图13-1　队列

这个接口并没有说明队列是如何实现的。队列通常有两种实现方式：一种是使用循环数组；另一种是使用链表（见图13-2）。

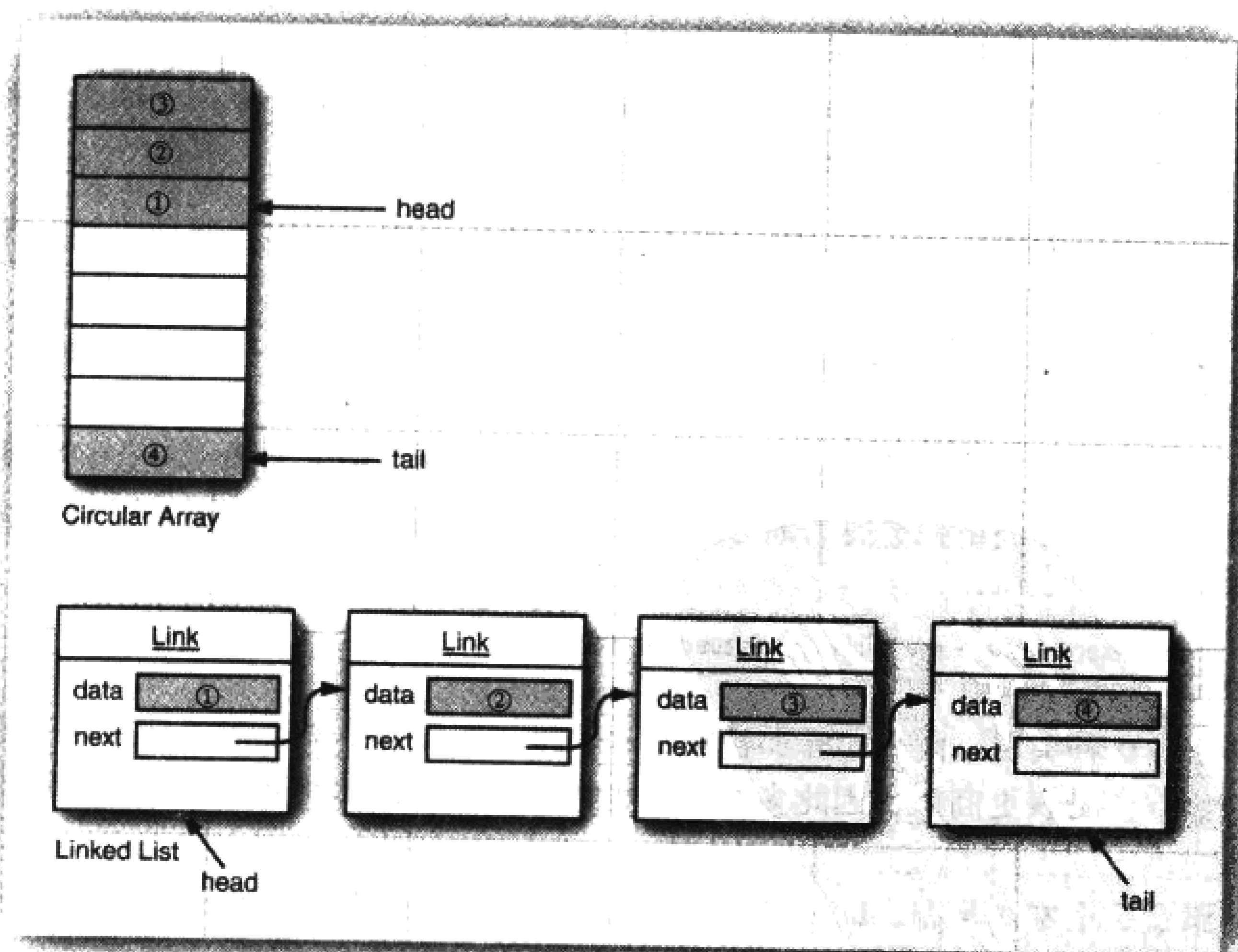


图13-2　队列的实现

注释：从Java SE 5.0开始，集合类是带有类型参数的泛型类。有关泛型类的更多信息，请参看第12章。

每一个实现都可以通过一个实现了Queue接口的类表示。

```
class CircularArrayQueue<E> implements Queue<E> // not an actual library class
{
   CircularArrayQueue(int capacity) { . . . }
   public void add(E element) { . . . }
   public E remove() { . . . }
   public int size() { . . . }
   private E[] elements;
   private int head;
   private int tail;
}

class LinkedListQueue<E> implements Queue<E>  // not an actual library class
{
   LinkedListQueue() { . . . }
   public void add(E element) { . . . }
   public E remove() { . . . }
   public int size() { . . . }

   private Link head;
   private Link tail;
}
```

✔ **注释**：实际上，Java类库没有名为CircularArrayQueue和LinkedListQueue的类。这里，只是以这些类作为示例，解释一下集合接口与实现在概念上的不同。如果需要一个循环数组队列，就可以使用Java SE 6中引入的ArrayDeque类。如果需要一个链表队列，就直接使用LinkedList类，这个类实现了Queue接口。

当在程序中使用队列时，一旦构建了集合就不需要知道究竟使用了哪种实现。因此，只有在构建集合对象时，使用具体的类才有意义。可以使用接口类型存放集合的引用。

```
Queue<Customer> expressLane = new CircularArrayQueue<Customer>(100);
expressLane.add(new Customer("Harry"));
```

利用这种方式，一旦改变了想法，可以轻松地使用另外一种不同的实现。只需要对程序的一个地方做出修改，即调用构造器的地方。如果最终觉得LinkedListQueue是个更好的选择，就将代码修改为：

```
Queue<Customer> expressLane = new LinkedListQueue<Customer>();
expressLane.add(new Customer("Harry"));
```

为什么选择这种实现，而不选择那种实现呢？接口本身并不能说明哪种实现的效率究竟如何。循环数组要比链表更高效，因此多数人优先选择循环数组。然而，通常这样做也需要付出一定的代价。

循环数组是一个有界集合，即容量有限。如果程序中要收集的对象数量没有上限，就最好使用链表来实现。

在研究API文档时，会发现另外一组名字以Abstract开头的类，例如，AbstractQueue。这些类是为类库实现者而设计的。如果想要实现自己的队列类（也许不太可能），会发现扩展AbstractQueue类要比实现Queue接口中的所有方法轻松得多。

13.1.2 Java类库中的集合接口和迭代器接口

在Java类库中，集合类的基本接口是Collection接口。

这个接口有两个基本方法：

```
public interface Collection<E>
{
   boolean add(E element);
   Iterator<E> iterator();
   . . .
}
```

除了这两个方法之外，还有几个方法，将在稍后介绍。

add方法用于向集合中添加元素。如果添加元素确实改变了集合就返回true，如果集合没有发生变化就返回false。例如，如果试图向集中添加一个对象，而这个对象在集中已经存在，这个添加请求就没有实效，因为集中不允许有重复的对象。

iterator方法用于返回一个实现了Iterator接口的对象。可以使用这个迭代器对象依次访问集合中的元素。

1. 迭代器

Iterator接口包含3个方法：

```
public interface Iterator<E>
{
   E next();
   boolean hasNext();
   void remove();
}
```

通过反复调用next方法，可以逐个访问集合中的每个元素。但是，如果到达了集合的末尾，next方法将抛出一个NoSuchElementException。因此，需要在调用next之前调用hasNext方法。如果迭代器对象还有多个供访问的元素，这个方法就返回true。如果想要查看集合中的所有元素，就请求一个迭代器，并在hasNext返回true时反复地调用next方法。例如：

```
Collection<String> c = . . .;
Iterator<String> iter = c.iterator();
while (iter.hasNext())
{
   String element = iter.next();
   do something with element
}
```

从Java SE 5.0起，这个循环可以采用一种更优雅的缩写方式。用“for each”循环可以更加简练地表示同样的循环操作：

```
for (String element : c)
{
   do something with element
}
```

编译器简单地将“for each”循环翻译为带有迭代器的循环。

“for each”循环可以与任何实现了Iterable接口的对象一起工作，这个接口只包含一个方法：

```
public interface Iterable<E>
{
   Iterator<E> iterator();
}
```

Collection接口扩展了Iterable接口。因此，对于标准类库中的任何集合都可以使用“for each”循环。

元素被访问的顺序取决于集合类型。如果对ArrayList进行迭代，迭代器将从索引0开始，每迭代一次，索引值加1。然而，如果访问HashSet中的元素，每个元素将会按照某种随机的次序出现。虽然可以确定在迭代过程中能够遍历到集合中的所有元素，但却无法预知元素被访问的次序。这对于计算总和或统计符合某个条件的元素个数这类与顺序无关的操作来说，并不是什么问题。

> **注释**：编程老手会注意到：Iterator接口的next和hasNext方法与Enumeration接口的nextElement和hasMoreElements方法的作用一样。Java集合类库的设计者可以选择使用Enumeration接口。但是，他们不喜欢这个接口累赘的方法名，于是引入了具有较短方法名的新接口。

Java集合类库中的迭代器与其他类库中的迭代器在概念上有着重要的区别。在传统的集合类库中，例如，C++的标准模版库，迭代器是根据数组索引建模的。如果给定这样一个迭代器，就可以查看指定位置上的元素，就像知道数组索引i就可以查看数组元素a[i]一样。不需要查找元素，就可以将迭代器向前移动一个位置。这与不需要执行查找操作就可以通过i++将数组索引向前移动一样。但是，Java迭代器并不是这样操作的。查找操作与位置变更是紧密相连的。查找一个元素的惟一方法是调用next，而在执行查找操作的同时，迭代器的位置随之向前移动。

因此，应该将Java迭代器认为是位于两个元素之间。当调用next时，迭代器就越过下一个元素，并返回刚刚越过的那个元素的引用（见图13-3）。

> **注释**：这里还有一个有用的类推。可以将Iterator.next 与InputStream.read看作为等效的。从数据流中读取一个字节，就会自动地“消耗掉” 这个字节。下一次调用read将会消耗并返回输入的下一个字节。用同样的方式，反复地调用next就可以读取集合中所有元素。

2. 删除元素

Iterator接口的remove方法将会删除上次调用next方法时返回的元素。在大多数情况下，在决定删除某个元素之前应该先看一下这个元素是很具有实际意义的。然而，如果想要删除指定位置上的元素，仍然需要越过这个元素。下面是如何删除字符串集合中第一个元素的方法：

```
Iterator<String> it = c.iterator();
it.next(); // skip over the first element
it.remove(); // now remove it
```

更重要的是，对next方法和remove方法的调用具有互相依赖性。如果调用remove之前没有调用next将是不合法的。如果这样做，将会抛出一个IllegalStateException异常。

如果想删除两个相邻的元素，不能直接地这样调用：

```
it.remove();
it.remove(); // Error!
```

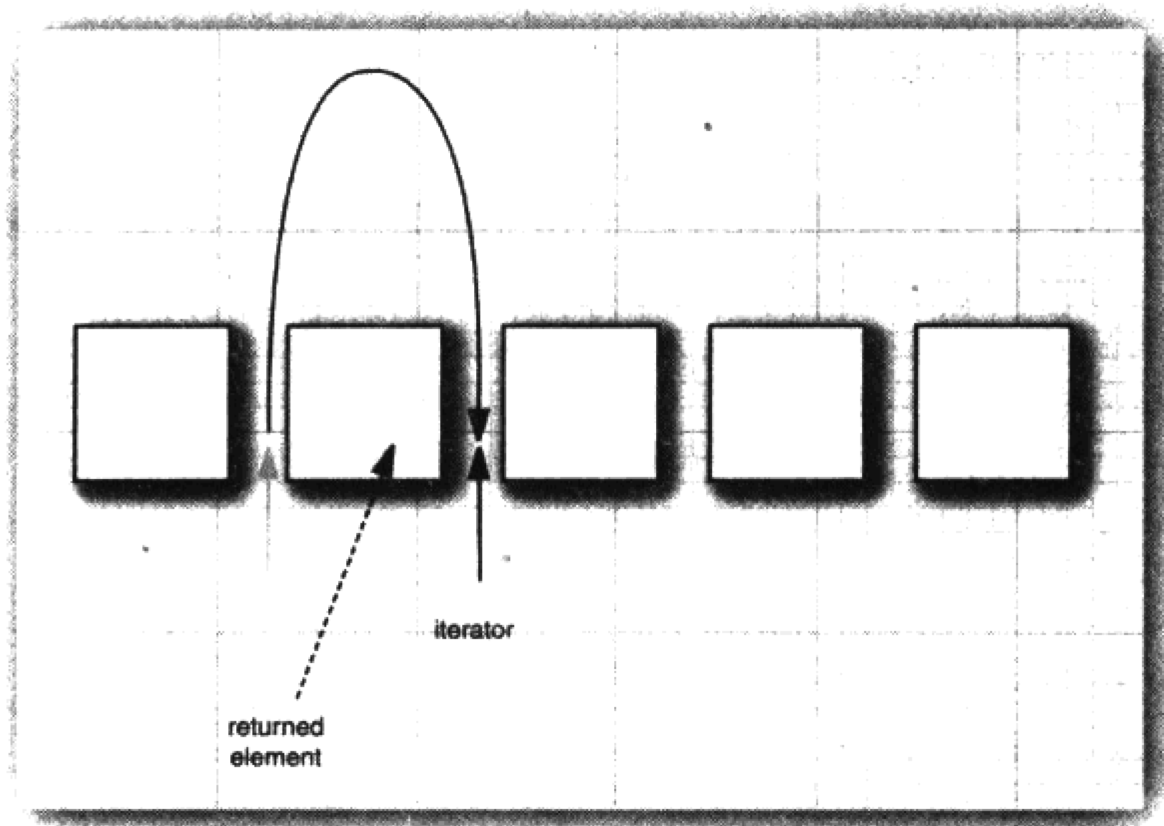


图13-3　向前移动迭代器

相反地，必须先调用next越过将要删除的元素。

```
it.remove();
it.next();
it.remove(); // Ok
```

3. 泛型实用方法

由于Collection与Iterator都是泛型接口，可以编写操作任何集合类型的实用方法。例如，下面是一个检测任意集合是否包含指定元素的泛型方法：

```
public static <E> boolean contains(Collection<E> c, Object obj)
{
   for (E element : c)
      if (element.equals(obj))
         return true;
   return false;
}
```

Java类库的设计者认为：这些实用方法中的某些方法非常有用，应该将它们提供给用户使用。这样，类库的使用者就不必自己重新构建这些方法了。contains就是这样一个实用方法。

事实上，Collectio 接口声明了很多有用的方法，所有的实现类都必须提供这些方法。下面列举了其中的一部分：

```
int size()
boolean isEmpty()
boolean contains(Object obj)
boolean containsAll(Collection<?> c)
boolean equals(Object other)
```

```
boolean addAll(Collection<? extends E> from)
boolean remove(Object obj)
boolean removeAll(Collection<?> c)
void clear()
boolean retainAll(Collection<?> c)
Object[] toArray()
<T> T[] toArray(T[] arrayToFill)
```

在这些方法中，有许多方法的功能非常明确，不需要过多的解释。在本节尾部的API注释中可以找到有关它们的完整文档说明。

当然，如果实现Collection接口的每一个类都要提供如此多的例行方法将是一件很烦人的事情。为了能够让实现者更容易地实现这个接口，Java类库提供了一个类AbstractCollection，它将基础方法size和iterator抽象化了，但是在此提供了例行方法。例如：

```
public abstract class AbstractCollection<E>
   implements Collection<E>
{
   . . .
   public abstract Iterator<E> iterator();

   public boolean contains(Object obj)
   {
      for (E element : c) // calls iterator()
         if (element.equals(obj))
            return = true;
      return false;
   }
   . . .
}
```

此时，一个具体的集合类可以扩展AbstractCollection类了。现在要由具体的集合类提供iterator方法，而contains方法已由AbstractCollection超类提供了。然而，如果子类有更加有效的方式实现contains方法，也可以由子类提供，就这点而言，没有什么限制。

对于类框架来说，这是一个很好的设计。集合类的用户可以使用泛型接口中一组更加丰富的方法，而实际的数据结构实现者并没有需要实现所有例行方法的负担。

API java.util.Collection<E> 1.2

- `Iterator<E> iterator()`

  返回一个用于访问集合中每个元素的迭代器。

- `int size()`

  返回当前存储在集合中的元素个数。

- `boolean isEmpty()`

  如果集合中没有元素，返回true。

- `boolean contains(Object obj)`

  如果集合中包含了一个与obj 相等的对象，返回true。

- `boolean containsAll(Collection<?> other)`

如果这个集合包含other集合中的所有元素，返回true。

- `boolean add(Object element)`

  将一个元素添加到集合中。如果由于这个调用改变了集合，返回true。

- `boolean addAll(Collection<? extends E> other)`

  将other集合中的所有元素添加到这个集合。如果由于这个调用改变了集合，返回true。

- `boolean remove(Object obj)`

  从这个集合中删除等于 obj 的对象。如果有匹配的对象被删除，返回true。

- `boolean removeAll(Collection<?> other)`

  从这个集合中删除other集合中存在的所有元素。如果由于这个调用改变了集合，返回true。

- `void clear()`

  从这个集合中删除所有的元素。

- `boolean retainAll(Collection<?> other)`

  从这个集合中删除所有与other集合中的元素不同的元素。如果由于这个调用改变了集合，返回true。

- `Object[] toArray()`

  返回这个集合的对象数组。

- `<T> T[] toArray(T[] arrayToFill)`

  返回这个集合的对象数组。如果arrayToFill足够大，就将集合中的元素填入这个数组中。剩余空间填补null；否则，分配一个新数组，其成员类型与arrayToFill的成员类型相同，其长度等于集合的大小，并添入集合元素。



`java.util.Iterator<E>` 1.2

- `boolean hasNext()`

  如果存在可访问的元素，返回true。

- `E next()`

  返回将要访问的下一个对象。如果已经到达了集合的尾部，将抛出一个NoSuchElement Exception。

- `void remove()`

  删除上次访问的对象。这个方法必须紧跟在访问一个元素之后执行。如果上次访问之后，集合已经发生了变化，这个方法将抛出一个IllegalStateException。

13.2 具体的集合

这里并不打算更加详细地介绍所有的接口，但我们认为先介绍一下Java类库提供的具体数据结构还是很有用途的。透彻地介绍了人们想使用的类之后，再回过头研究一些抽象的概念，看一看集合框架组织这些类的方式。表13-1展示了Java类库中的集合，并简要描述了每个集合类的用途（鉴于简单起见，省略了将在第14章中介绍的线程安全集合）。在表13-1中，除了以Map结尾的类之外，其他类都实现了Collection 接口。而以Map结尾的类实现了Map接口。有关

Map接口的内容将在稍后介绍。

表13-1 Java库中的具体集合

集合类型	描　述
ArrayList	一种可以动态增长和缩减的索引序列
LinkedList	一种可以在任何位置进行高效地插入和删除操作的有序序列
ArrayDeque	一种用循环数组实现的双端队列
HashSet	一种没有重复元素的无序集合
TreeSet	一种有序集
EnumSet	一种包含枚举类型值的集
LinkedHashSet	一种可以记住元素插入次序的集
PriorityQueue	一种允许高效删除最小元素的集合
HashMap	一种存储键/值关联的数据结构
TreeMap	一种键值有序排列的映射表
EnumMap	一种键值属于枚举类型的映射表
LinkedHashMap	一种可以记住键/值项添加次序的映射表
WeakHashMap	一种其值无用武之地后可以被垃圾回收器回收的映射表
IdentityHashMap	一种用==，而不是用equals比较键值的映射表

13.2.1 链表

在本书中，有很多示例已经使用了数组以及动态的ArrayList类。然而，数组和数组列表都有一个重大的缺陷。这就是从数组的中间位置删除一个元素要付出很大的代价，其原因是数组中处于被删除元素之后的所有元素都要向数组的前端移动（见图13-4）。在数组中间的位置上插入一个元素也是如此。

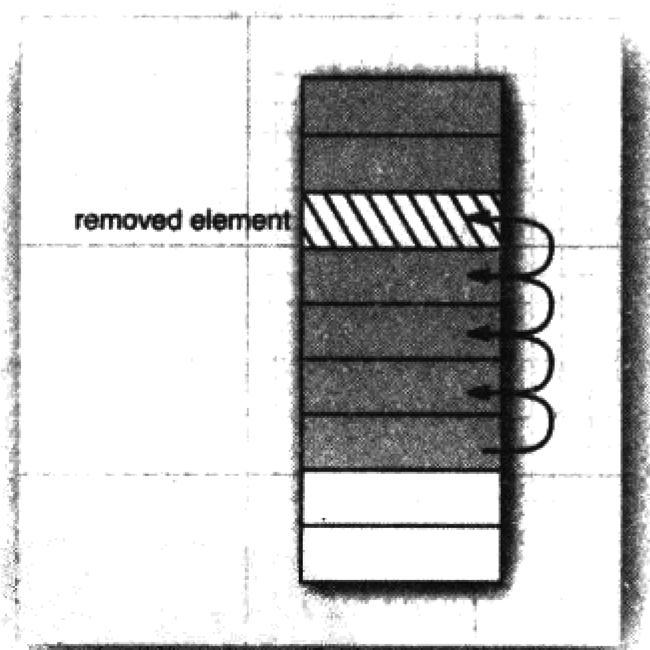


图13-4 从数组中删除一个元素

另外一个大家非常熟悉的数据结构——链表（linked list）解决了这个问题。尽管数组在连续的存储位置上存放对象引用，但链表却将每个对象存放在独立的结点中。每个结点还存放着序列中下一个结点的引用。在Java程序设计语言中，所有链表实际上都是双向链接的（doubly linked）——即每个结点还存放着指向前驱结点的引用（见图13-5）。

从链表中间删除一个元素是一个很轻松的操作，即需要对被删除元素附近的结点更新一下即可（见图13-6）。

你也许曾经在数据结构课程中学习过如何实现链表的操作。在链表中添加或删除元素时，绕来绕去的指针可能已经给人们留下了极坏的印象。如果真是如此的话，就会为Java集合类库提供一个类LinkedList而感到拍手称快。

在下面的代码示例中，先添加3个元素，然后再将第2个元素删除：

```
List<String> staff = new LinkedList<String>(); // LinkedList implements List
```

```
staff.add("Amy");
staff.add("Bob");
staff.add("Carl");
Iterator iter = staff.iterator();
String first = iter.next(); // visit first element
String second = iter.next(); // visit second element
iter.remove(); // remove last visited element
```

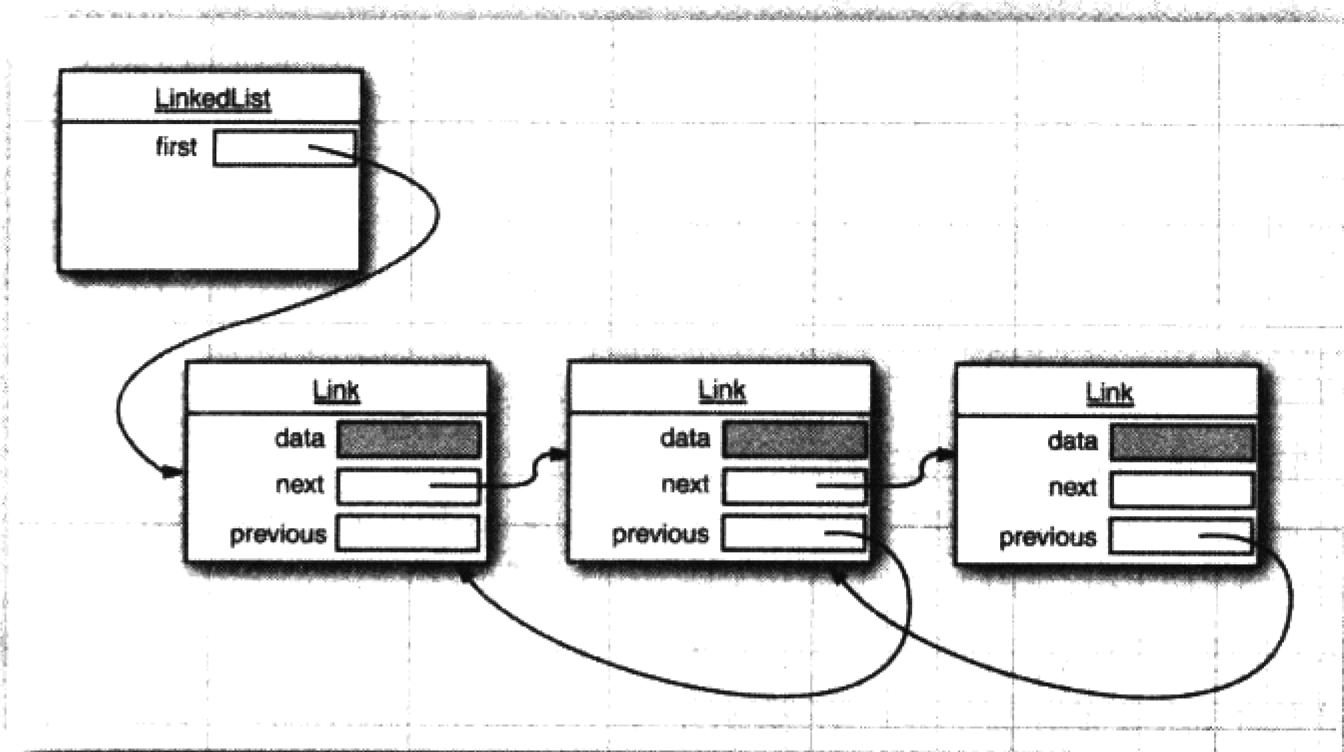


图13-5 双向链表

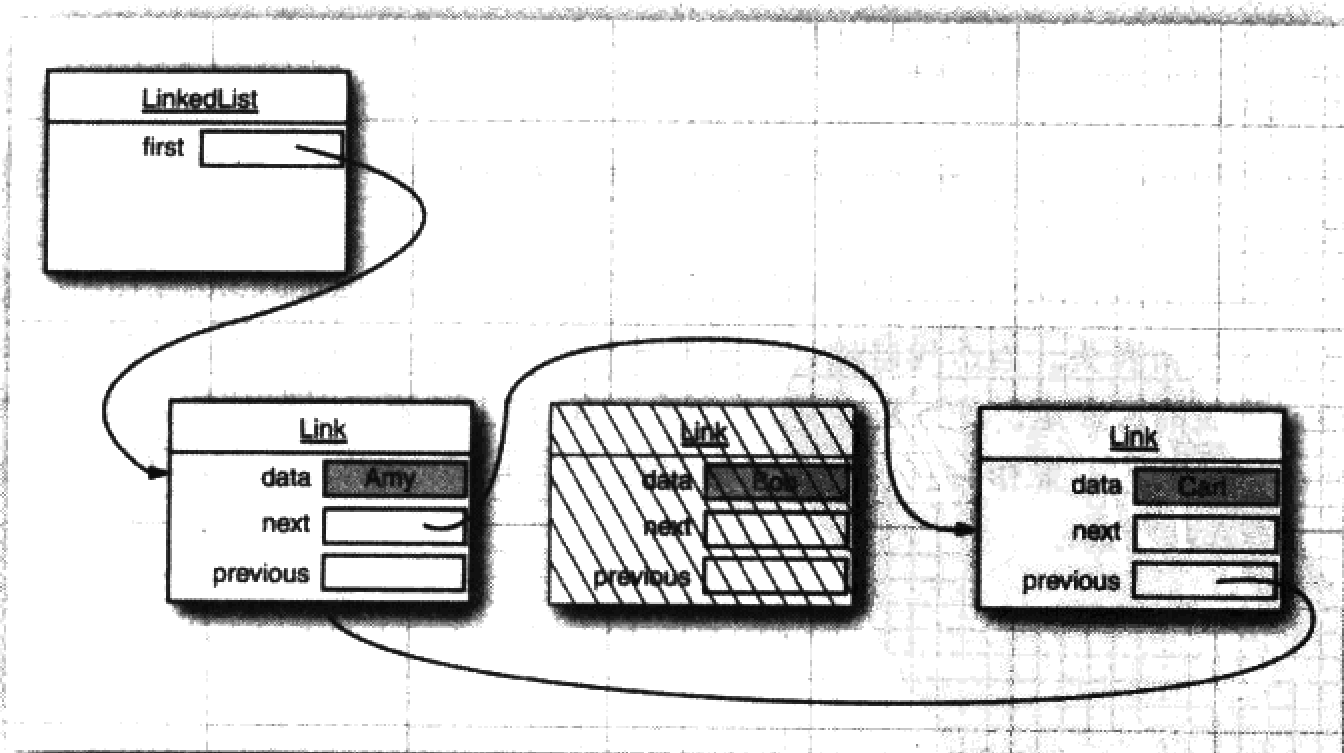


图13-6 从链表中删除一个元素

但是，链表与泛型集合之间有一个重要的区别。链表是一个有序集合（ordered collection），

每个对象的位置十分重要。LinkedList.add 方法将对象添加到链表的尾部。但是，常常需要将元素添加到链表的中间。由于迭代器是描述集合中位置的，所以这种依赖于位置的add方法将由迭代器负责。只有对自然有序的集合使用迭代器添加元素才有实际意义。例如，下一节将要讨论的集（set）类型，其中的元素完全无序。因此，在Iterator 接口中就没有add方法。相反地，集合类库提供了子接口ListIterator，其中包含add方法：

```
interface ListIterator<E> extends Iterator<E>
{
   void add(E element);
   . . .
}
```

与Collection.add不同，这个方法不返回boolean类型的值，它假定添加操作总会改变链表。另外，ListIterator接口有两个方法，可以用来反向遍历链表。

```
E previous()
boolean hasPrevious()
```

与next方法一样，previous方法返回越过的对象。

LinkedList类的listIterator方法返回一个实现了ListIterator接口的迭代器对象。

```
ListIterator<String> iter = staff.listIterator();
```

Add方法在迭代器位置之前添加一个新对象。例如，下面的代码将越过链表中的第一个元素，并在第二个元素之前添加“Juliet”（见图13-7）：

```
List<String> staff = new LinkedList<String>();
staff.add("Amy");
staff.add("Bob");
staff.add("Carl");
ListIterator<String> iter = staff.listIterator();
iter.next(); // skip past first element
iter.add("Juliet");
```

如果多次调用add方法，将按照提供的次序把元素添加到链表中。它们被依次添加到迭代器当前位置之前。

当用一个刚刚由Iterator方法返回，并且指向链表表头的迭代器调用add操作时，新添加的元素将变成列表的新表头。当迭代器越过链表的最后一个元素时（即hasNext返回false），添加的元素将变成列表的新表尾。如果链表有n个元素，有n+1个位置可以添加新元素。这些位置与迭代器的n+1个可能的位置相对应。例如，如果链表包含3个元素，A、B、C，就有4个位置（标有|）可以插入新元素：

```
|ABC
A|BC
AB|C
ABC|
```



注释：在用“光标”类比时要格外小心。remove操作与BACKSPACE键的工作方式不太一样。在调用next之后，remove方法确实与BACKSPACE键一样删除了迭代器左侧的元素。但是，如果调用previous就会将右侧的元素删除掉，并且不能在同一行中调用两次remove。

add方法只依赖于迭代器的位置，而remove方法依赖于迭代器的状态。

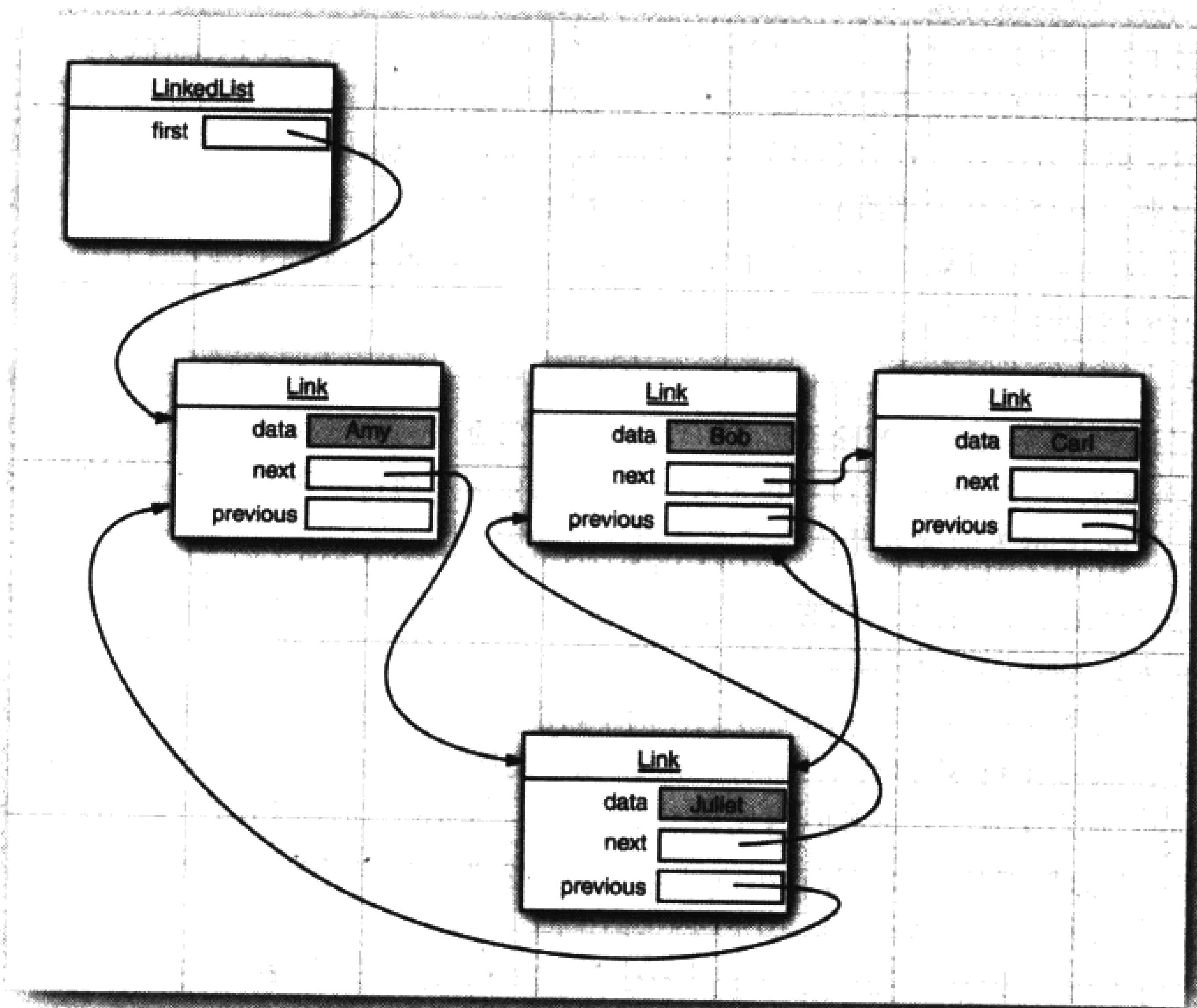


图13-7 将一个元素添加到链表中

最后需要说明，set方法用一个新元素取代调用next或previous方法返回的上一个元素。例如，下面的代码将用一个新值取代链表的第一个元素：

```
ListIterator<String> iter = list.listIterator();
String oldValue = iter.next(); // returns first element
iter.set(newValue); // sets first element to newValue
```

可以想像，如果在某个迭代器修改集合时，另一个迭代器对其进行遍历，一定会出现混乱的状况。例如，一个迭代器指向另一个迭代器刚刚删除的元素前面，现在这个迭代器就是无效的，并且不应该再使用。链表迭代器的设计使它能够检测到这种修改。如果迭代器发现它的集合被另一个迭代器修改了，或是被该集合自身的方法修改了，就会抛出一个Concurrent ModificationException异常。例如，看一看下面这段代码：

```
List<String> list = . . .;
ListIterator<String> iter1 = list.listIterator();
ListIterator<String> iter2 = list.listIterator();
iter1.next();
iter1.remove();
iter2.next(); // throws ConcurrentModificationException
```

由于iter2检测出这个链表被从外部修改了，所以对iter2.next的调用抛出了一个 Concurrent ModificationException异常。

为了避免发生并发修改的异常，请遵循下述简单规则：可以根据需要给容器附加许多的迭代器，但是这些迭代器只能读取列表。另外，再单独附加一个既能读又能写的迭代器。

有一种简单的方法可以检测到并发修改的问题。集合可以跟踪改写操作（诸如添加或删除元素）的次数。每个迭代器都维护一个独立的计数值。在每个迭代器方法的开始处检查自己改写操作的计数值是否与集合的改写操作计数值一致。如果不一致，抛出一个Concurrent ModificationException异常。

注释：对于并发修改列表的检测有一个奇怪的例外。链表只负责跟踪对列表的结构性修改，例如，添加元素、删除元素。set操作不被视为结构性修改。可以将多个迭代器附加给一个链表，所有的迭代器都调用set方法对现有结点的内容进行修改。在本章后面所介绍的Collections类的许多算法都需要使用这个功能。

现在已经介绍了LinkedList类的各种基本方法。可以使用ListIterator类从前后两个方向遍历链表中的元素，并可以添加、删除元素。

在上一节已经看到，Collection接口中声明了许多用于对链表操作的有用方法。其中大部分方法都是在LinkedList类的超类AbstractCollection中实现的。例如，toString方法调用了所有元素的toString，并产生了一个很长的格式为 [A,B, C]的字符串。这为调试工作提供了便利。可以使用contains方法检测某个元素是否出现在链表中。例如，如果链表中包含一个等于“Harry”的字符串，调用staff.contains(“Harry”)后将会返回true。

在Java类库中，还提供了许多在理论上存在一定争议的方法。链表不支持快速地随机访问。如果要查看链表中第n个元素，就必须从头开始，越过n－1个元素。没有捷径可走。鉴于这个原因，在程序需要采用整数索引访问元素时，程序员通常不选用链表。

尽管如此，LinkedList类还是提供了一个用来访问某个特定元素的get方法：

```
LinkedList<String> list = . . .;
String obj = list.get(n);
```

当然，这个方法的效率并不太高。如果发现自己正在使用这个方法，说明有可能对于所要解决的问题使用了错误的数据结构。

绝对不应该使用这种让人误解的随机访问方法来遍历链表。下面这段代码的效率极低：

```
for (int i = 0; i < list.size(); i++)
   do something with list.get(i);
```

每次查找一个元素都要从列表的头部重新开始搜索。LinkedList对象根本不做任何缓存位置信息的操作。

注释：get方法做了微小的优化：如果索引大于size() / 2就从列表尾端开始搜索元素。

列表迭代器接口还有一个方法，可以告之当前位置的索引。实际上，从概念上讲，由于Java迭代器指向两个元素之间的位置，所以可以同时产生两个索引：nextIndex方法返回下一次

调用next方法时返回元素的整数索引；previousIndex方法返回下一次调用previous方法时返回元素的整数索引。当然，这个索引只比nextIndex返回的索引值小1。这两个方法的效率非常高，这是因为迭代器保持着当前位置的计数值。最后需要说一下，如果有一个整数索引n，list.listIterator(n) 将返回一个迭代器，这个迭代器指向索引为n的元素前面的位置。也就是说，调用next与调用list.get(n)会产生同一个元素，只是获得这个迭代器的效率比较低。

如果链表中只有很少几个元素，就完全没有必要为get方法和set方法的开销而烦恼。但是，为什么要优先使用链表呢？使用链表的惟一理由是尽可能地减少在列表中间插入或删除元素所付出的代价。如果列表只有少数几个元素，就完全可以使用ArrayList。

我们建议避免使用以整数索引表示链表中位置的所有方法。如果需要对集合进行随机访问，就使用数组或ArrayList, 而不要使用链表。

例13-1中的程序使用的就是链表。它简单地创建了两个链表，将它们合并在一起，然后从第二个链表中每间隔一个元素删除一个元素，最后测试removeAll方法。建议跟踪一下程序流程，并要特别注意迭代器。从这里会发现绘制一个下面这样的迭代器位置示意图是非常有用的：

```
|ACE   |BDFG
A|CE   |BDFG
AB|CE  B|DFG
. . .
```

注意调用

```
System.out.println(a);
```

通过调用AbstractCollection类中的toString方法打印出链表a中的所有元素。

例13-1 LinkedListTest.java

```java
import java.util.*;

/**
 * This program demonstrates operations on linked lists.
 * @version 1.10 2004-08-02
 * @author Cay Horstmann
 */
public class LinkedListTest
{
   public static void main(String[] args)
   {
      List<String> a = new LinkedList<String>();
      a.add("Amy");
      a.add("Carl");
      a.add("Erica");

      List<String> b = new LinkedList<String>();
      b.add("Bob");
      b.add("Doug");
      b.add("Frances");
      b.add("Gloria");

      // merge the words from b into a

```

```
      ListIterator<String> aIter = a.listIterator();
      Iterator<String> bIter = b.iterator();

      while (bIter.hasNext())
      {
         if (aIter.hasNext()) aIter.next();
         aIter.add(bIter.next());
      }

      System.out.println(a);

      // remove every second word from b

      bIter = b.iterator();
      while (bIter.hasNext())
      {
         bIter.next(); // skip one element
         if (bIter.hasNext())
         {
            bIter.next(); // skip next element
            bIter.remove(); // remove that element
         }
      }

      System.out.println(b);

      // bulk operation: remove all words in b from a

      a.removeAll(b);

      System.out.println(a);
   }
}
```

API java.util.List<E> 1.2

- `ListIterator<E> listIterator()`

  返回一个列表迭代器，以便用来访问列表中的元素。

- `ListIterator<E> listIterator(int index)`

  返回一个列表迭代器，以便用来访问列表中的元素，这个元素是第一次调用next返回的给定索引的元素。

- `void add(int i, E element)`

  在给定位置添加一个元素。

- `void addAll(int i, Collection<? extends E> elements)`

  将某个集合中的所有元素添加到给定位置。

- `E remove(int i)`

  删除给定位置的元素并返回这个元素。

- `E get(int i)`

  获取给定位置的元素。

- `E set(int i, E element)`

  用新元素取代给定位置的元素，并返回原来那个元素。
- `int indexOf(Object element)`

  返回与指定元素相等的元素在列表中第一次出现的位置，如果没有这样的元素将返回 -1。
- `int lastIndexOf(Object element)`

  返回与指定元素相等的元素在列表中最后一次出现的位置，如果没有这样的元素将返回 -1。

API `java.util.ListIterator<E>` 1.2

- `void add(E newElement)`

  在当前位置前添加一个元素。
- `void set(E newElement)`

  用新元素取代next或previous上次访问的元素。如果在next或previous上次调用之后列表结构被修改了，将抛出一个IllegalStateException 异常。
- `boolean hasPrevious()`

  当反向迭代列表时，还有可供访问的元素，返回true。
- `E previous()`

  返回前一个对象。如果已经到达了列表的头部，就抛出一个 NoSuchElementException异常。
- `int nextIndex()`

  返回下一次调用next方法时将返回的元素索引。
- `int previousIndex()`

  返回下一次调用previous 方法时将返回的元素索引。

API `java.util.LinkedList<E>` 1.2

- `LinkedList()`

  构造一个空链表。
- `LinkedList(Collection<? extends E> elements)`

  构造一个链表，并将集合中所有的元素添加到这个链表中。
- `void addFirst(E element)`
- `void addLast(E element)`

  将某个元素添加到列表的头部或尾部。
- `E getFirst()`
- `E getLast()`

  返回列表头部或尾部的元素。
- `E removeFirst()`
- `E removeLast()`

  删除并返回列表头部或尾部的元素。

13.2.2 数组列表

在上一节中，介绍了List接口和实现了这个接口的LinkedList类。List接口用于描述一个有序集合，并且集合中每个元素的位置十分重要。有两种访问元素的协议：一种是用迭代器，另一种是用get和set方法随机地访问每个元素。后者不适用于链表，但对数组却很有用。 集合类库提供了一种大家熟悉的ArrayList类，这个类也实现了List接口。ArrayList封装了一个动态再分配的对象数组。



注释：对于一个经验丰富的Java程序员来说，在需要动态数组时，可能会使用Vector类。为什么要用ArrayList取代Vector呢？原因很简单：Vector类的所有方法都是同步的。可以由两个线程安全地访问一个Vector对象。但是，如果由一个线程访问Vector，代码要在同步操作上耗费大量的时间。这种情况还是很常见的。而ArrayList方法不是同步的，因此，建议在不需要同步时使用ArrayList，而不要使用Vector。

13.2.3 散列集

链表和数组可以按照人们的意愿排列元素的次序。但是，如果想要查看某个指定的元素，却又忘记了它的位置，就需要访问所有元素，直到找到为止。如果集合中包含的元素很多，将会消耗很多时间。如果不在意元素的顺序，可以有几种能够快速查找元素的数据结构。其缺点是无法控制元素出现的次序。它们将按照有利于其操作目的的原则组织数据。

有一种众所周知的数据结构，可以快速地查找所需要的对象，这就是*散列表*（hash table）。散列表为每个对象计算一个整数，称为*散列码*（hash code）。散列码是由对象的实例域产生的一个整数。更准确地说，具有不同数据域的对象将产生不同的散列码。表13-2列出了几个散列码的示例，它们是由String类的hashCode方法产生的。

表13-2 由hashCode函数导出的散列码

串	散列码
"Lee"	76268
"lee"	107020
"eel"	100300

如果自定义类，就要负责实现这个类的hashCode方法。有关hashCode方法的详细内容请参看第5章。注意，自己实现的hashCode方法应该与equals方法兼容，即如果a.equals(b)为true，a与b必须具有相同的散列码。

现在，最重要的问题是散列码要能够快速地计算出来，并且这个计算只与要散列的对象状态有关，与散列表中的其他对象无关。

在Java中，散列表用链表数组实现。每个列表被称为*桶*（bucket）（参看图13-8）。要想查找表中对象的位置，就要先计算它的散列码，然后与桶的总数取余，所得到的结果就是保存这个元素的桶的索引。例如，如果某个对象的散列码为76268，并且有128个桶，对象应该保存在第108号桶中（76268除以128余108）。或许会很幸运，在这个桶中没有其他元素，此时将元素直接插入到桶中就可以了。当然，有时候会遇到桶被占满的情况，这也是不可避免的。这种现象被称为*散列冲突*（hash collision）。这时，需要用新对象与桶中的所有对象进行比较，查看这个对象是否已经存在。如果散列码是合理且随机分布的，桶的数目也足够大，需要比较的次数

就会很少。

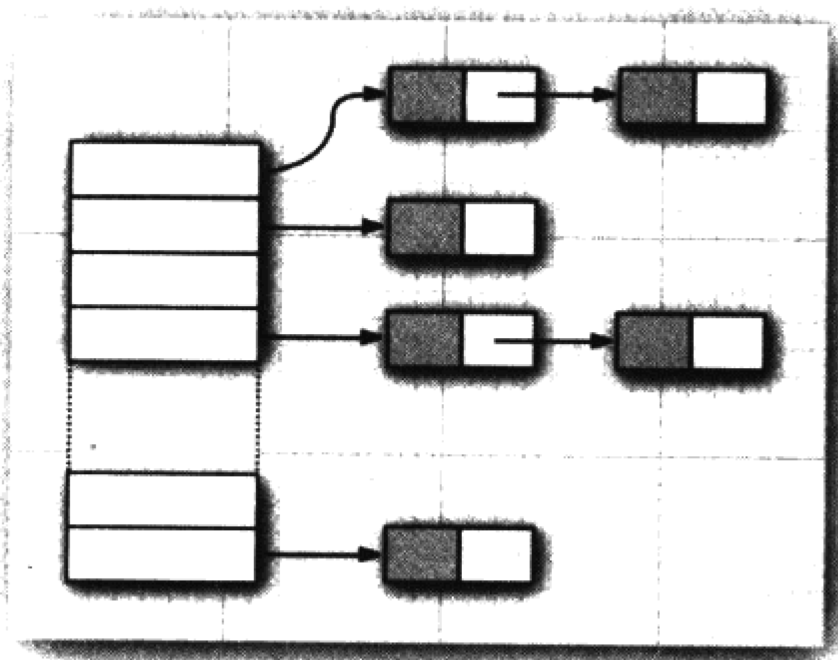


图13-8 散列表

如果想更多地控制散列表的运行性能，就要指定一个初始的桶数。桶数是指用于收集具有相同散列值的桶的数目。如果要插入到散列表中的元素太多，就会增加冲突的可能性，降低运行性能。

如果大致知道最终会有多少个元素要插入到散列表中，就可以设置桶数。通常，将桶数设置为预计元素个数的75%~150%。有些研究人员认为：尽管还没有确凿的证据，但最好将桶数设置为一个素数，以防键的集聚。标准类库使用的桶数是2的幂，默认值为16（为表大小提供的任何值都将被自动地转换为2的下一个幂）。

当然，并不是总能够知道需要存储多少个元素的，也有可能最初的估计过低。如果散列表太满，就需要*再散列*（rehashed）。如果要对散列表再散列，就需要创建一个桶数更多的表，并将所有元素插入到这个新表中，然后丢弃原来的表。*装填因子*（load factor）决定何时对散列表进行再散列。例如，如果装填因子为0.75（默认值），而表中超过75%的位置已经填入元素，这个表就会用双倍的桶数自动地进行再散列。对于大多数应用程序来说，装填因子为75%是比较合理的。

散列表可以用于实现几个重要的数据结构。其中最简单的是set类型。set是没有重复元素的元素集合。set的add方法首先在集中查找要添加的对象，如果不存在，就将这个对象添加进去。

Java集合类库提供了一个HashSet类，它实现了基于散列表的集。可以用add方法添加元素。contains方法已经被重新定义，用来快速地查看是否某个元素已经出现在集中。它只在某个桶中查找元素，而不必查看集合中的所有元素。

散列集迭代器将依次访问所有的桶。由于散列将元素分散在表的各个位置上，所以访问它们的顺序几乎是随机的。只有不关心集合中元素的顺序时才应该使用HashSet。

本节末尾的示例程序（例13-2）将从System.in读取单词，然后将它们添加到集中，最后，再打印出集中的所有单词。例如，可以将*Alice in Wonderland*（可以从http://www.gutenberg.net找到）的文本输入到这个程序中，并通过下列命令运行：

```
java SetTest < alice30.txt
```

这个程序将读取输入的所有单词，并且将它们添加到散列集中。然后遍历散列集中的不同单词，最后打印出单词的数量（*Alice in Wonderland*共有5909个不同的单词，包括开头的版权声明)。单词以随机的顺序出现。

警告： *在更改集中的元素时要格外小心。如果元素的散列码发生了改变，元素在数据结构中的位置也会发生变化。*

例13-2 SetTest.java

```
import java.util.*;

/**
 * This program uses a set to print all unique words in System.in.
 * @version 1.10 2003-08-02
 * @author Cay Horstmann
 */
public class SetTest
{
   public static void main(String[] args)
   {
      Set<String> words = new HashSet<String>(); // HashSet implements Set
      long totalTime = 0;

      Scanner in = new Scanner(System.in);
      while (in.hasNext())
      {
         String word = in.next();
         long callTime = System.currentTimeMillis();
         words.add(word);
         callTime = System.currentTimeMillis() - callTime;
         totalTime += callTime;
      }

      Iterator<String> iter = words.iterator();
      for (int i = 1; i <= 20; i++)
         System.out.println(iter.next());
      System.out.println(". . .");
      System.out.println(words.size() + " distinct words. " + totalTime + " milliseconds.");
   }
}
```

API java.util.HashSet<E> 1.2

- `HashSet()`

  构造一个空散列表。
- `HashSet(Collection<? extends E> elements)`

  构造一个散列集，并将集合中的所有元素添加到这个散列集中。
- `HashSet(int initialCapacity)`

  构造一个空的具有指定容量（桶数）的散列集。

• HashSet(int initialCapacity, float loadFactor)

构造一个具有指定容量和装填因子（一个0.0~1.0之间的数值，确定散列表填充的百分比，当大于这个百分比时，散列表进行再散列）的空散列集。

API java.lang.Object 1.0

• int hashCode()

返回这个对象的散列码。散列码可以是任何整数，包括正数或负数。equals和hashCode的定义必须兼容，即如果x.equals(y)为true，x.hashCode()必须等于y.hashCode()。

13.2.4 树集

TreeSet类与散列集十分类似，不过，它比散列集有所改进。树集是一个有序集合（sorted collection）。可以以任意顺序将元素插入到集合中。在对集合进行遍历时，每个值将自动地按照排序后的顺序呈现。例如，假设插入3个字符串，然后访问添加的所有元素。

```
SortedSet<String> sorter = new TreeSet<String>(); // TreeSet implements SortedSet
sorter.add("Bob");
sorter.add("Amy");
sorter.add("Carl");
for (String s : sorter) System.println(s);
```

这时，每个值将按照顺序打印出来：Amy Bob Carl。正如TreeSet类名所示，排序是用树结构完成的（当前实现使用的是红黑树（red-black tree）。有关红黑树的详细介绍请参看《Introduction to Algorithms》，作者是Thomas Cormen、Charles Leiserson、Ronald Rivest和Clifford Stein [The MIT Press, 2001]）[⊖] 每次将一个元素添加到树中时，都被放置在正确的排序位置上。因此，迭代器总是以排好序的顺序访问每个元素。

将一个元素添加到树中要比添加到散列表中慢，但是，与将元素添加到数组或链表的正确位置上相比还是快很多的。如果树中包含n个元素，查找新元素的正确位置平均需要$\log_2 n$次比较。例如，如果一棵树包含了1000个元素，添加一个新元素大约需要比较10次。

因此，将一个元素添加到TreeSet中要比添加到HashSet中慢。请参看表13-3。不过，TreeSet可以自动地对元素进行排序。

表13-3 将元素添加到散列集和树集

文 档	单词总数	不同的单词个数	HashSet	TreeSet
Alice in Wonderland	28195	5909	5秒	7秒
The Count of Monte Cristo	466300	37545	75秒	98秒

API java.util.TreeSet<E> 1.2

• TreeSet()

构造一个空树集。

⊖ 本书中文版《算法导论》已由机械工业出版社出版。——编辑注

- `TreeSet(Collection<? extends E> elements)`

  构造一个树集，并将集合中的所有元素添加到树集中。

13.2.5 对象的比较

TreeSet如何知道希望元素怎样排列呢？在默认情况时，树集假定插入的元素实现了Comparable接口。这个接口定义了一个方法：

```
public interface Comparable<T>
{
   int compareTo(T other);
}
```

如果a与b相等，调用a.compareTo(b)一定返回0；如果排序后a位于b之前，则返回负值；如果a位于b之后，则返回正值。具体返回什么值并不重要，关键是符号（>0、0或<0）。有些标准的Java平台类实现了Comparable接口，例如，String类。这个类的compareTo方法依据字典序（有时称为词典序）对字符串进行比较。

如果要插入自定义的对象，就必须通过实现Comparable接口自定义排列顺序。在Object 类中，没有提供任何compareTo接口的默认实现。

例如，下面的代码展示了如何用部件编号对Item对象进行排序：

```
class Item implements Comparable<Item>
{
   public int compareTo(Item other)
   {
      return partNumber - other.partNumber;
   }
   . . .
}
```

如果对两个正整数进行比较，就像上面示例中的部件编号，就可以直接地返回它们的差。如果第一项[⊖]位于第二项的前面，就返回负值；如果部件编号相同就返回0；否则返回正值。

警告：只有整数在一个足够小的范围内，才可以使用这个技巧。如果x是一个较大的正整数，y是一个较大的负整数，x－y有可能会溢出。

然而，使用Comparable接口定义排列排序显然有其局限性。对于一个给定的类，只能够实现这个接口一次。如果在一个集合中需要按照部件编号进行排序，在另一个集合中却要按照描述信息进行排序，该怎么办呢？另外，如果需要对一个类的对象进行排序，而这个类的创建者又没有费心实现Comparable接口，又该怎么办呢？

在这种情况下，可以通过将Comparator对象传递给TreeSet构造器来告诉树集使用不同的比较方法。Comparator接口声明了一个带有两个显式参数的compare方法：

```
public interface Comparator<T>
{
   int compare(T a, T b);
}
```

⊖ 指产品。——译者注

与compareTo方法一样，如果a位于b之前compare方法则返回负值；如果a和b相等则返回0，否则返回正值。

如果按照描述信息进行排序，就直接定义一个实现Comparator接口的类：

```
class ItemComparator implements Comparator<Item>
{
   public int compare(Item a, Item b)
   {
      String descrA = a.getDescription();
      String descrB = b.getDescription();
      return descrA.compareTo(descrB);
   }
}
```

然后将这个类的对象传递给树集的构造器：

```
ItemComparator comp = new ItemComparator();
SortedSet<Item> sortByDescription = new TreeSet<Item>(comp);
```

如果构造了一棵带比较器的树，就可以在需要比较两个元素时使用这个对象。

注意，这个比较器没有任何数据。它只是比较方法的持有器。有时将这种对象称为函数对象（function object）。函数对象通常被定义为“瞬时的”，即匿名内部类的实例：

```
SortedSet<Item> sortByDescription = new TreeSet<Item>(new
   Comparator<Item>()
   {
      public int compare(Item a, Item b)
      {
         String descrA = a.getDescription();
         String descrB = b.getDescription();
         return descrA.compareTo(descrB);
      }
   });
```

注释：实际上，Comparator<T>接口声明了两个方法：compare和equals。当然，每一个类都有一个 equals 方法；因此，为这个接口声明再添加一个equels方法似乎没有太大好处。API文档解释说，不需要覆盖equals方法，但这样做可能会在某些情况下提高性能。例如，如果从另一个集合添加元素，这个由使用相同比较器的另外一个集添加元素，TreeSet类中的addAll方法的效率会更高。

回头看一看表13-3可能会疑虑：是否总是应该用树集取代散列集。毕竟，添加一个元素所花费的时间看上去并不很长，而且元素是自动排序的。到底应该怎样做将取决于所要收集的数据。如果不需要对数据进行排序，就没有必要付出排序的开销。更重要的是，对于某些数据来说，对其排序要比散列函数更加困难。散列函数只是将对象适当地打乱存放，而比较却要精确地判别每个对象。

要想具体地了解它们之间的差异，还需要研究一个收集矩形集的任务。如果使用 TreeSet，就需要提供Comparator<Rectangle>。如何比较两个矩形呢？比较面积吗？这行不通。可能会有两个不同的矩形，它们的坐标不同，但面积却相同。树的排序必须是整体排序。也就是说，任意两个元素必须是可比的，并且只有在两个元素相等时结果才为0。确实，有一种矩形的排序

（按照坐标的词典顺序排列）方式，但它的计算很牵强且很繁琐。相反地，Rectangle类已经定义了散列函数，它直接对坐标进行散列。

> **注释**：从Java SE 6起，TreeSet类实现了NavigableSet接口。这个接口增加了几个便于定位元素以及反向遍历的方法。详细信息请参看API注释。

在例13-3的程序中创建了两个Item对象的树集。第一个按照部件编号排序，这是Item对象的默认顺序。第二个通过使用一个定制的比较器来按照描述信息排序。

例13-3 TreeSetTest.java

```
/**
   @version 1.10 2004-08-02
   @author Cay Horstmann
*/

import java.util.*;

/**
   This program sorts a set of items by comparing
   their descriptions.
*/
public class TreeSetTest
{
   public static void main(String[] args)
   {
      SortedSet<Item> parts = new TreeSet<Item>();
      parts.add(new Item("Toaster", 1234));
      parts.add(new Item("Widget", 4562));
      parts.add(new Item("Modem", 9912));
      System.out.println(parts);

      SortedSet<Item> sortByDescription = new TreeSet<Item>(new
         Comparator<Item>()
         {
            public int compare(Item a, Item b)
            {
               String descrA = a.getDescription();
               String descrB = b.getDescription();
               return descrA.compareTo(descrB);
            }
         });

      sortByDescription.addAll(parts);
      System.out.println(sortByDescription);
   }
}

/**
   An item with a description and a part number.
*/
class Item implements Comparable<Item>
{
   /**
      Constructs an item.
```

```
      @param aDescription the item's description
      @param aPartNumber the item's part number
   */
   public Item(String aDescription, int aPartNumber)
   {
      description = aDescription;
      partNumber = aPartNumber;
   }

   /**
      Gets the description of this item.
      @return the description
   */
   public String getDescription()
   {
      return description;
   }

   public String toString()
   {
      return "[descripion=" + description
         + ", partNumber=" + partNumber + "]";
   }

   public boolean equals(Object otherObject)
   {
      if (this == otherObject) return true;
      if (otherObject == null) return false;
      if (getClass() != otherObject.getClass()) return false;
      Item other = (Item) otherObject;
      return description.equals(other.description)
         && partNumber == other.partNumber;
   }

   public int hashCode()
   {
      return 13 * description.hashCode() + 17 * partNumber;
   }

   public int compareTo(Item other)
   {
      return partNumber - other.partNumber;
   }

   private String description;
   private int partNumber;
}
```

API java.lang.Comparable<T> 1.2

- `int compareTo(T other)`

  将这个对象（this）与另一个对象（other）进行比较，如果this位于other之前则返回负值；如果两个对象在排列顺序中处于相同的位置则返回0，如果this位于other之后则返回正值。

API java.util.Comparator<T> 1.2

- int compare(T a, T b)

  将两个对象进行比较，如果a位于b之前则返回负值；如果两个对象在排列顺序中处于相同的位置则返回0；如果a位于b之后则返回正值。

API java.util.SortedSet<E> 1.2

- Comparator<? super E> comparator()

  返回用于对元素进行排序的比较器。如果元素用Comparable 接口的compareTo方法进行比较则返回null。
- E first()
- E last()

  返回有序集中的最小元素或最大元素。

API java.util.NavigableSet<E> 6

- E higher(E value)
- E lower(E value)

  返回大于value的最小元素或小于value的最大元素，如果没有这样的元素则返回null。
- E ceiling(E value)
- E floor(E value)

  返回大于等于value的最小元素或小于等于value的最大元素，如果没有这样的元素则返回null。
- E pollFirst()
- E pollLast

  删除并返回这个集中的最大元素或最小元素，这个集为空时返回null。
- Iterator<E> descendingIterator()

  返回一个按照递减顺序遍历集中元素的迭代器。

API java.util.TreeSet<E> 1.2

- TreeSet()

  构造一个用于排列Comparable对象的树集。
- TreeSet(Comparator<? super E> c)

  构造一个树集，并使用指定的比较器对其中的元素进行排序。
- TreeSet(SortedSet<? extends E> elements)

  构造一个树集，将有序集中的所有元素添加到这个树集中，并使用与给定集相同的元素比较器。

13.2.6 队列与双端队列

前面已经讨论过，队列可以让人们有效地在尾部添加一个元素，在头部删除一个元素。有两个端头的队列，即*双端队列*，可以让人们有效地在头部和尾部同时添加或删除元素。不支持在队列中间添加元素。在Java SE 6中引入了Deque接口，并由ArrayDeque和LinkedList类实现。这两个类都提供了双端队列，而且在必要时可以增加队列的长度。在第14章将会看到有限队列和有限双端队列。

API java.util.Queue<E> 5.0

- `boolean add(E element)`
- `boolean offer(E element)`

  如果队列没有满，将给定的元素添加到这个双端队列的尾部并返回true。如果队列满了，第一个方法将抛出一个IllegalStateException，而第二个方法返回false。
- `E remove()`
- `E poll()`

  假如队列不空，删除并返回这个队列头部的元素。如果队列是空的，第一个方法抛出NoSuchElementException，而第二个方法返回null。
- `E element()`
- `E peek()`

  如果队列不空，返回这个队列头部的元素，但不删除。如果队列空，第一个方法将抛出一个NoSuchElementException，而第二个方法返回null。

API java.util.Deque<E> 6

- `void addFirst(E element)`
- `void addLast(E element)`
- `boolean offerFirst(E element)`
- `boolean offerLast(E element)`

  将给定的对象添加到双端队列的头部或尾部。如果队列满了，前面两个方法将抛出一个IllegalStateException，而后面两个方法返回false。
- `E removeFirst()`
- `E removeLast()`
- `E pollFirst()`
- `E pollLast()`

  如果队列不空，删除并返回队列头部的元素。如果队列为空，前面两个方法将抛出一个NoSuchElementException，而后面两个方法返回null。
- `E getFirst()`
- `E getLast()`
- `E peekFirst()`

- E peekLast()

  如果队列非空，返回队列头部的元素，但不删除。如果队列空，前面两个方法将抛出一个NoSuchElementException，而后面两个方法返回null。

API java.util.ArrayDeque<E> 6

- ArrayDeque()
- ArrayDeque(int initialCapacity)

  用初始容量16或给定的初始容量构造一个无限双端队列。

13.2.7 优先级队列

优先级队列（priority queue）中的元素可以按照任意的顺序插入，却总是按照排序的顺序进行检索。也就是说，无论何时调用remove方法，总会获得当前优先级队列中最小的元素。然而，优先级队列并没有对所有的元素进行排序。如果用迭代的方式处理这些元素，并不需要对它们进行排序。优先级队列使用了一个优雅且高效的数据结构，称为堆（heap）。堆是一个可以自我调整的二叉树，对树执行添加（add）和删除（remore）操作，可以让最小的元素移动到根，而不必花费时间对元素进行排序。

与TreeSet一样，一个优先级队列既可以保存实现了Comparable接口的类对象，也可以保存在构造器中提供比较器的对象。

使用优先级队列的典型示例是任务调度。每一个任务有一个优先级，任务以随机顺序添加到队列中。每当启动一个新的任务时，都将优先级最高的任务从队列中删除（由于习惯上将1设为“最高”优先级，所以会将最小的元素删除）。

例13-4显示了一个正在运行的优先级队列。与TreeSet中的迭代不同，这里的迭代并不是按照元素的排列顺序访问的。而删除却总是删掉剩余元素中优先级数最小的那个元素。

例13-4 PriorityQueueTest.java

```java
import java.util.*;

/**
 * This program demonstrates the use of a priority queue.
 * @version 1.00 2004-08-03
 * @author Cay Horstmann
 */
public class PriorityQueueTest
{
   public static void main(String[] args)
   {
      PriorityQueue<GregorianCalendar> pq = new PriorityQueue<GregorianCalendar>();
      pq.add(new GregorianCalendar(1906, Calendar.DECEMBER, 9)); // G. Hopper
      pq.add(new GregorianCalendar(1815, Calendar.DECEMBER, 10)); // A. Lovelace
      pq.add(new GregorianCalendar(1903, Calendar.DECEMBER, 3)); // J. von Neumann
      pq.add(new GregorianCalendar(1910, Calendar.JUNE, 22)); // K. Zuse

      System.out.println("Iterating over elements...");
      for (GregorianCalendar date : pq)
```

```
        System.out.println(date.get(Calendar.YEAR));
      System.out.println("Removing elements...");
      while (!pq.isEmpty())
         System.out.println(pq.remove().get(Calendar.YEAR));
   }
}
```

API java.util.PriorityQueue 5.0

- `PriorityQueue()`
- `PriorityQueue(int initialCapacity)`

  构造一个用于存放Comparable对象的优先级队列。
- `PriorityQueue(int initialCapacity, Comparator<? super E> c)`

  构造一个优先级队列，并用指定的比较器对元素进行排序。

13.2.8 映射表

集是一个集合，它可以快速地查找现有的元素。但是，要查看一个元素，需要有要查找元素的精确副本。这不是一种非常通用的查找方式。通常，我们知道某些键的信息，并想要查找与之对应的元素。*映射表*（map）数据结构就是为此设计的。映射表用来存放键/值对。如果提供了键，就能够查找到值。例如，有一张关于员工信息的记录表，键为员工ID，值为Employee对象。

Java类库为映射表提供了两个通用的实现：HashMap和TreeMap。 这两个类都实现了Map接口。

散列映射表对键进行散列，树映射表用键的整体顺序对元素进行排序，并将其组织成搜索树。散列或比较函数*只能作用于键*。与键关联的值不能进行散列或比较。

应该选择散列映射表还是树映射表呢？与集一样，散列稍微快一些，如果不需要按照排列顺序访问键，就最好选择散列。

下列代码将为存储的员工信息建立一个散列映射表：

```
Map<String, Employee> staff = new HashMap<String, Employee>(); // HashMap implements Map
Employee harry = new Employee("Harry Hacker");
staff.put("987-98-9996", harry);
. . .
```

每当往映射表中添加对象时，必须同时提供一个键。在这里，键是一个字符串，对应的值是Employee对象。

要想检索一个对象，必须提供（因而，必须记住）一个键。

```
String s = "987-98-9996";
e = staff.get(s); // gets harry
```

如果在映射表中没有与给定键对应的信息，get将返回null。

键必须是惟一的。不能对同一个键存放两个值。如果对同一个键两次调用put方法，第二个值就会取代第一个值。实际上，put将返回用这个键参数存储的上一个值。

remove方法用于从映射表中删除给定键对应的元素。size方法用于返回映射表中的元素数。

集合框架并没有将映射表本身视为一个集合（其他的数据结构框架则将映射表视为对（pairs）的集合，或者视为用键作为索引的值的集合）。然而，可以获得映射表的视图，这是一组实现了Collection接口对象，或者它的子接口的视图。

有3个视图，它们分别是：键集、值集合（不是集）和键/值对集。键与键/值对形成了一个集，这是因为在映射表中一个键只能有一个副本。下列方法将返回这3个视图（条目集的元素是静态内部类Map.Entry的对象）。

```
Set<K> keySet()
Collection<K> values()
Set<Map.Entry<K, V>> entrySet()
```

注意，keySet既不是HashSet，也不是TreeSet，而是实现了Set接口的某个其他类的对象。Set接口扩展了Collection接口。因此，可以与使用任何集合一样使用keySet。

例如，可以枚举映射表中的所有键：

```
Set<String> keys = map.keySet();
for (String key : keys)
{
   do something with key
}
```

提示： 如果想要同时查看键与值，就可以通过枚举各个条目（entries）查看，以避免对值进行查找。可以使用下面这段代码框架：

```
for (Map.Entry<String, Employee> entry : staff.entrySet())
{
   String key = entry.getKey();
   Employee value = entry.getValue();
   do something with key, value
}
```

如果调用迭代器的remove方法，实际上就从映射表中删除了键以及对应的值。但是，不能将元素添加到键集的视图中。如果只添加键而不添加值是毫无意义的。如果试图调用add方法，将会抛出一个UnsupportedOperationException异常。条目集视图也有同样的限制，不过，从概念上讲，添加新的键/值对是有意义的。

例13-5显示了映射表的操作过程。首先将键/值对添加到映射表中。然后，从映射表中删除一个键，同时与之对应的值也被删除掉了。接下来，修改与某一个键对应的值，并调用get方法查看这个值。最后，对条目集进行迭代。

例13-5 MapTest.java

```
import java.util.*;

/**
 * This program demonstrates the use of a map with key type String and value type Employee.
 * @version 1.10 2004-08-02
 * @author Cay Horstmann
 */
```

```java
public class MapTest
{
   public static void main(String[] args)
   {
      Map<String, Employee> staff = new HashMap<String, Employee>();
      staff.put("144-25-5464", new Employee("Amy Lee"));
      staff.put("567-24-2546", new Employee("Harry Hacker"));
      staff.put("157-62-7935", new Employee("Gary Cooper"));
      staff.put("456-62-5527", new Employee("Francesca Cruz"));

      // print all entries

      System.out.println(staff);

      // remove an entry

      staff.remove("567-24-2546");

      // replace an entry

      staff.put("456-62-5527", new Employee("Francesca Miller"));

      // look up a value

      System.out.println(staff.get("157-62-7935"));

      // iterate through all entries

      for (Map.Entry<String, Employee> entry : staff.entrySet())
      {
         String key = entry.getKey();
         Employee value = entry.getValue();
         System.out.println("key=" + key + ", value=" + value);
      }
   }
}

/**
 * A minimalist employee class for testing purposes.
 */
class Employee
{
   /**
    * Constructs an employee with $0 salary.
    * @param n the employee name
    */
   public Employee(String n)
   {
      name = n;
      salary = 0;
   }

   public String toString()
   {
      return "[name=" + name + ", salary=" + salary + "]";
   }
```

```
    
    private String name;
    private double salary;
}
```

API java.util.Map<K, V> 1.2

- V get(K key)

  获取与键对应的值；返回与键对应的对象，如果在映射表中没有这个对象则返回null。键可以为null。
- V put(K key, V value)

  将键与对应的值关系插入到映射表中。如果这个键已经存在，新的对象将取代与这个键对应的旧对象。这个方法将返回键对应的旧值。如果这个键以前没有出现过则返回null。键可以为null，但值不能为null。
- void putAll(Map<? extends K, ? extends V> entries)

  将给定映射表中的所有条目添加到这个映射表中。
- boolean containsKey(Object key)

  如果在映射表中已经有这个键，返回true。
- boolean containsValue(Object value)

  如果映射表中已经有这个值，返回true。
- Set<Map.Entry<K, V>> entrySet()

  返回Map.Entry对象的集视图，即映射表中的键/值对。可以从这个集中删除元素，同时也从映射表中删除了它们。但是，不能添加任何元素。
- Set<K> keySet()

  返回映射表中所有键的集视图。可以从这个集中删除元素，同时也从映射表中删除了它们。但是，不能添加任何元素。
- Collection<V> values()

  返回映射表中所有值的集合视图。可以从这个集中删除元素，同时也从映射表中删除了它们。但是，不能添加任何元素。

API java.util.Map.Entry<K, V> 1.2

- K getKey()
- V getValue()

  返回这个条目的键或值。
- V setValue(V newValue)

  设置在映射表中与新值对应的值，并返回旧值。

API java.util.HashMap<K, V> 1.2

- HashMap()

- `HashMap(int initialCapacity)`
- `HashMap(int initialCapacity, float loadFactor)`

  用给定的容量和装填因子构造一个空散列映射表（装填因子是一个0.0~1.0之间的数值。这个数值决定散列表填充的百分比。一旦到了这个比例，就要将其再散列到更大的表中）。默认的装填因子是0.75 。

API java.util.TreeMap<K,V> 1.2

- `TreeMap(Comparator<? super K> c)`

  构造一个树映射表，并使用一个指定的比较器对键进行排序。
- `TreeMap(Map<? extends K, ? extends V> entries)`

  构造一个树映射表，并将某个映射表中的所有条目添加到树映射表中。
- `TreeMap(SortedMap<? extends K, ? extends V> entries)`

  构造一个树映射表，将某个有序映射表中的所有条目添加到树映射表中，并使用与给定的有序映射表相同的比较器。

API java.util.SortedMap<K, V> 1.2

- `Comparator<? super K> comparator()`

  返回对键进行排序的比较器。如果键是用Comparable接口的compareTo方法进行比较的，返回null。
- `K firstKey()`
- `K lastKey()`

  返回映射表中最小元素和最大元素。

13.2.9 专用集与映射表类

在集合类库中，有几个专用的映射表类，本节对它们做一下简要地介绍。

1. *弱散列映射表*

设计WeakHashMap类是为了解决一个有趣的问题。如果有一个值，对应的键已经不再使用了，将会出现什么情况呢？假定对某个键的最后一次引用已经消亡，不再有任何途径引用这个值的对象了。但是，由于在程序中的任何部分没有再出现这个键，所以，这个键/值对无法从映射表中删除。为什么垃圾回收器不能够删除它呢？难道删除无用的对象不是垃圾回收器的工作吗？

遗憾的是，事情没有这样简单。垃圾回收器跟踪*活动的*对象。只要映射表对象是活动的，其中的所有桶也是活动的，它们不能被回收。因此，需要由程序负责从长期存活的映射表中删除那些无用的值。或者使用WeakHashMap完成这件事情。当对键的惟一引用来自散列表条目时，这一数据结构将与垃圾回收器协同工作一起删除键/值对。

下面是这种机制的内部运行情况。WeakHashMap使用*弱引用*（weak references）保存键。WeakReference对象将引用保存到另外一个对象中，在这里，就是散列表键。对于这种类型的对象，垃圾回收器用一种特有的方式进行处理。通常，如果垃圾回收器发现某个特定的对象已

经没有他人引用了，就将其回收。然而，如果某个对象只能由WeakReference引用，垃圾回收器仍然回收它，但要将引用这个对象的弱引用放入队列中。WeakHashMap将周期性地检查队列，以便找出新添加的弱引用。一个弱引用进入队列意味着这个键不再被他人使用，并且已经被收集起来。于是，WeakHashMap将删除对应的条目。

2. 链接散列集和链接映射表

Java SE 1.4增加了两个类：LinkedHashSet和LinkedHashMap，用来记住插入元素项的顺序。这样就可以避免在散列表中的项从表面上看是随机排列的。当条目插入到表中时，就会并入到双向链表中（见图13-9）。

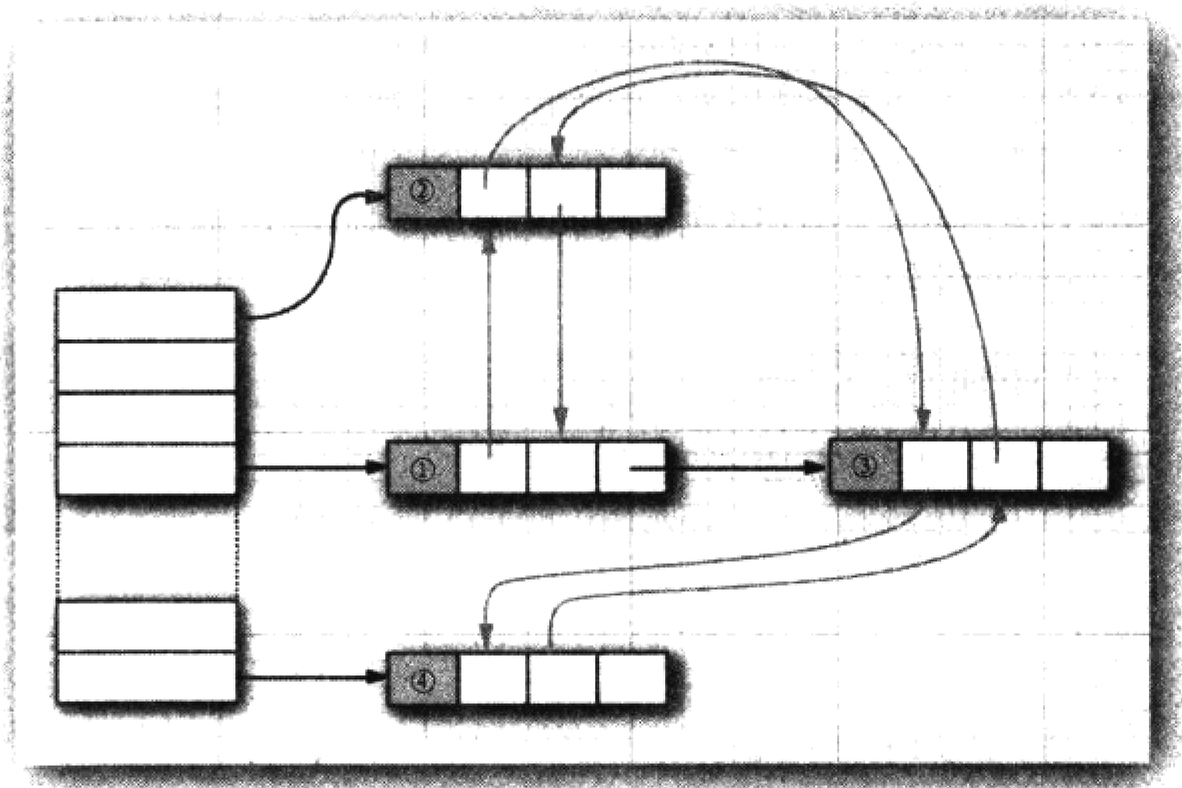


图13-9 链接散列表

例如，在例13-5中包含下列映射表插入的处理：

```
Map staff = new LinkedHashMap();
staff.put("144-25-5464", new Employee("Amy Lee"));
staff.put("567-24-2546", new Employee("Harry Hacker"));
staff.put("157-62-7935", new Employee("Gary Cooper"));
staff.put("456-62-5527", new Employee("Francesca Cruz"));
```

然后，staff.keySet().iterator()以下面次序枚举键：

```
144-25-5464
567-24-2546
157-62-7935
456-62-5527
```

并且staff.values().iterator()以下列顺序枚举这些值：

```
Amy Lee
Harry Hacker
Gary Cooper
Francesca Cruz
```

链接散列映射表将用访问顺序，而不是插入顺序，对映射表条目进行迭代。每次调用get或put，受到影响的条目将从当前的位置删除，并放到条目链表的尾部（只有条目在链表中的位置会受影响，而散列表中的桶不会受影响。一个条目总位于与键散列码对应的桶中）。要项构造这样一个的散列映射表，请调用

```
LinkedHashMap<K, V>(initialCapacity, loadFactor, true)
```

访问顺序对于实现高速缓存的“最近最少使用”原则十分重要。例如，可能希望将访问频率高的元素放在内存中，而访问频率低的元素则从数据库中读取。当在表中找不到元素项且表又已经满时，可以将迭代器加入到表中，并将枚举的前几个元素删除掉。这些是近期最少使用的几个元素。

甚至可以让这一过程自动化。即构造一个LinkedHashMap的子类，然后覆盖下面这个方法：

```
protected boolean removeEldestEntry(Map.Entry<K, V> eldest)
```

每当方法返回true时，就添加一个新条目，从而导致删除eldest条目。例如，下面的高速缓存可以存放100个元素：

```
Map<K, V> cache = new
   LinkedHashMap<K, V>(128, 0.75F, true)
   {
      protected boolean removeEldestEntry(Map.Entry<K, V> eldest)
      {
         return size() > 100;
      }
   };
```

另外，还可以对eldest条目进行评估，以此决定是否应该将它删除。例如，可以检查与这个条目一起存在的时间戳。

3. 枚举集与枚举映射表

EnumSet是一个枚举类型元素集的高效实现。由于枚举类型只有有限个实例，所以EnumSet内部用位序列实现。如果对应的值在集中，则相应的位被置为1。

EnumSet类没有公共的构造器。可以使用静态工厂方法构造这个集：

```
enum Weekday { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY };
EnumSet<Weekday> always = EnumSet.allOf(Weekday.class);
EnumSet<Weekday> never = EnumSet.noneOf(Weekday.class);
EnumSet<Weekday> workday = EnumSet.range(Weekday.MONDAY, Weekday.FRIDAY);
EnumSet<Weekday> mwf = EnumSet.of(Weekday.MONDAY, Weekday.WEDNESDAY, Weekday.FRIDAY);
```

可以使用Set接口的常用方法来修改EnumSet。

EnumMap是一个键类型为枚举类型的映射表。它可以直接且高效地用一个值数组实现。在使用时，需要在构造器中指定键类型：

```
EnumMap<Weekday, Employee> personInCharge = new EnumMap<Weekday, Employee>(Weekday.class);
```

注释：在EnumSet的API文档中，将会看到E extends Enum<E>这样奇怪的类型参数。简单地说，它的意思是“E是一个枚举类型。”所有的枚举类型都扩展于泛型Enum类。例如，Weekday扩展于Enum<Weekday>。

4. 标识散列映射表

Java SE 1.4还为另外一个特殊目的增加了另一个类IdentityHashMap。在这个类中，键的散列值不是用hashCode函数计算的，而是用System.identityHashCode方法计算的。这是Object.hashCode方法根据对象的内存地址来计算散列码时所使用的方式。而且，在对两个对象进行比较时，IdentityHashMap类使用==，而不使用equals。

也就是说，不同的键对象，即使内容相同，也被视为是不同的对象。在实现对象遍历算法（如对象序列化）时，这个类非常有用，可以用来跟踪每个对象的遍历状况。

API java.util.WeakHashMap<K, V> 1.2

- WeakHashMap()
- WeakHashMap(int initialCapacity)
- WeakHashMap(int initialCapacity, float loadFactor)

  用给定的容量和填充因子构造一个空的散列映射表。

API java.util.LinkedHashSet<E> 1.4

- LinkedHashSet()
- LinkedHashSet(int initialCapacity)
- LinkedHashSet(int initialCapacity, float loadFactor)

  用给定的容量和填充因子构造一个空链接散列集。

API java.util.LinkedHashMap<K, V> 1.4

- LinkedHashMap()
- LinkedHashMap(int initialCapacity)
- LinkedHashMap(int initialCapacity, float loadFactor)
- LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)

  用给定的容量、填充因子和顺序构造一个空的链接散列映射表。
- protected boolean removeEldestEntry(Map.Entry<K, V> eldest)

  如果想删除eldest元素，并同时返回true，就应该覆盖这个方法。eldest参数是预期要删除的条目。这个方法将在条目添加到映射表中之后调用。其默认的实现将返回false。即在默认情况下，旧元素没有被删除。然而，可以重新定义这个方法，以便有选择地返回true。例如，如果最旧的条目符合一个条件，或者映射表超过了一定大小，则返回true。

API java.util.EnumSet<E extends Enum<E>> 5.0

- static <E extends Enum<E>> EnumSet<E> allOf(Class<E> enumType)

  返回一个包含给定枚举类型的所有值的集。
- static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> enumType)

  返回一个空集，并有足够的空间保存给定的枚举类型所有的值。

- static <E extends Enum<E>> EnumSet<E> range(E from, E to)

  返回一个包含from~to之间的所有值（包括两个边界元素）的集。
- static <E extends Enum<E>> EnumSet<E> of(E value)
- static <E extends Enum<E>> EnumSet<E> of(E value, E... values)

  返回包括给定值的集。

API java.util.EnumMap<K extends Enum<K>, V> 5.0

- EnumMap(Class<K> keyType)

  构造一个键为给定类型的空映射集。

API java.util.IdentityHashMap<K, V> 1.4

- IdentityHashMap()
- IdentityHashMap(int expectedMaxSize)

  构造一个空的标识散列映射集，其容量是大于1.5 * expectedMaxSize的2的最小次幂（expectedMaxSize的默认值是21）。

API java.lang.System 1.0

- static int identityHashCode(Object obj) 1.1

  返回Object.hashCode计算出来的相同散列码（根据对象的内存地址产生），即使obj所属的类已经重新定义了hashCode方法也是如此。

13.3 集合框架

框架（framework）是一个类的集，它奠定了创建高级功能的基础。框架包含很多超类，这些超类拥有非常有用的功能、策略和机制。框架使用者创建的子类可以扩展超类的功能，而不必重新创建这些基本的机制。例如，Swing就是一种用户界面的机制。

Java集合类库构成了集合类的框架。它为集合的实现者定义了大量的接口和抽象类（见图13-10），并且对其中的某些机制给予了描述，例如，迭代协议。正如前面几节所做的那样，可以使用这些集合类，而不必了解框架。但是，如果想要实现用于多种集合类型的泛型算法，或者是想要增加新的集合类型，了解一些框架的知识是很有帮助的。

集合有两个基本的接口：Collection和Map。可以使用下列方法向集合中插入元素：

```
boolean add(E element)
```

但是，由于映射表保存的是键/值对，所以可以使用put方法进行插入。

```
V put(K key, V value)
```

要想从集合中读取某个元素，就需要使用迭代器访问它们。然而，也可以用get方法从映射表读取值：

```
V get(K key)
```

List是一个有序集合（ordered collection）。元素可以添加到容器中某个特定的位置。将对

象放置在某个位置上可以采用两种方式：使用整数索引或使用列表迭代器。List接口定义了几个用于随机访问的方法：

```
void add(int index, E element)
E get(int index)
void remove(int index)
```

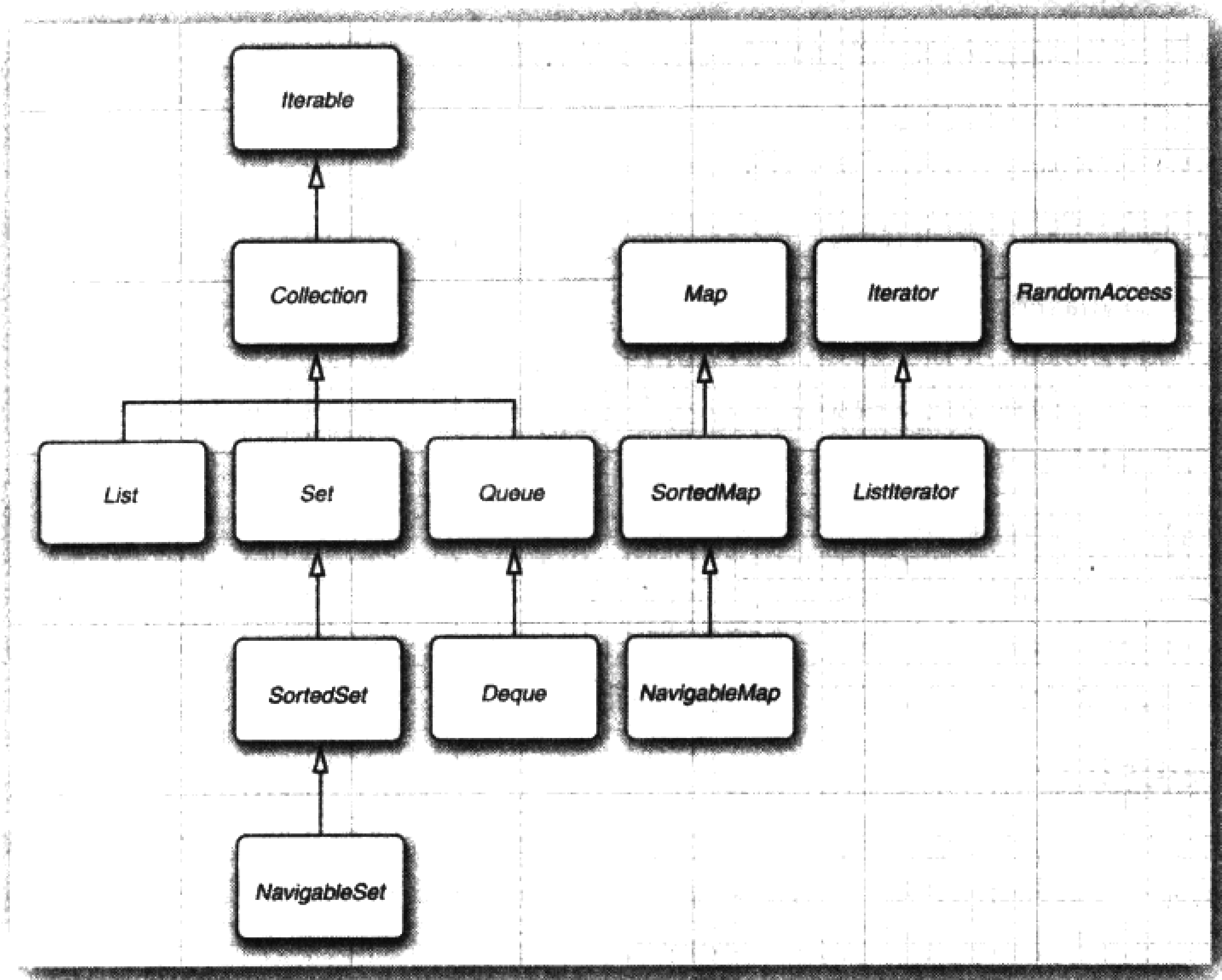


图13-10 集合框架的接口

如同前面所讨论的，List接口在提供这些随机访问方法时，并不管它们对某种特定的实现是否高效。为了避免执行成本较高的随机访问操作，Java SE 1.4引入了一个标记接口RandomAccess。这个接口没有任何方法，但可以用来检测一个特定的集合是否支持高效的随机访问：

```
if (c instanceof RandomAccess)
{
   use random access algorithm
}
else
{
   use sequential access algorithm
}
```

ArrayList类和Vector类都实现了RandomAccess接口。

> ✔ **注释**：从理论上看，有一个独立的Array接口，它扩展于List接口，并声明了随机访问方法是有很合理的。如果确实有这样一个独立的Array接口，那些需要随机访问的算法就可以使用Array参数，而且也不会无意中将它们应用于随机访问速度很慢的集合了。但是，集合框架的设计者没有选择去定义一个独立的接口，其原因是希望让类库中的接口数量保持在一个较少的水平。此外，不希望对程序员采取一种家长式的作风。这样可以自由地将链表传递给随机访问算法，只是需要清楚这样做会给性能带来什么不良的影响。

ListIterator接口定义了一个方法，用于将一个元素添加到迭代器所处位置的前面：

```
void add(E element)
```

要想获取和删除给定位置的元素，只需要调用Iterator接口中的next方法和remove方法即可。

Set接口与Collection接口是一样的，只是其方法的行为有着更加严谨的定义。集的add方法拒绝添加重复的元素。集的equals方法定义两个集相等的条件是它们包含相同的元素但顺序不必相同。hashCode方法定义应该保证具有相同元素的集将会得到相同的散列码。

既然方法签名是相同的，为什么还要建立一个独立的接口呢？从概念上讲，并不是所有集合都是集。建立Set接口后，可以让程序员编写仅接受集的方法。

SortedSet和SortedMap接口暴露了用于排序的比较器对象，并且定义的方法可以获得集合的子集视图。下一节将讨论这些视图。

最后，Java SE 6引入了接口NavigableSet和NavigableMap，其中包含了几个用于在有序集和映射表中查找和遍历的方法（从理论上讲，这几个方法已经包含在 SortedSet 和 SortedMap 的接口中）。TreeSet和TreeMap类实现了这几个接口。

现在，让我们将话题从接口转到实现接口的类上。前面已经讨论过，集合接口有大量的方法，这些方法可以通过更基本的方法加以实现。抽象类提供了许多这样的例行实现：

```
AbstractCollection
AbstractList
AbstractSequentialList
AbstractSet
AbstractQueue
AbstractMap
```

如果实现了自己的集合类，就可能要扩展上面某个类，以便可以选择例行操作的实现。

Java类库支持下面几种具体类：

```
LinkedList
ArrayList
ArrayDeque
HashSet
TreeSet
PriorityQueue
HashMap
TreeMap
```

图13-11展示了这些类之间的关系。

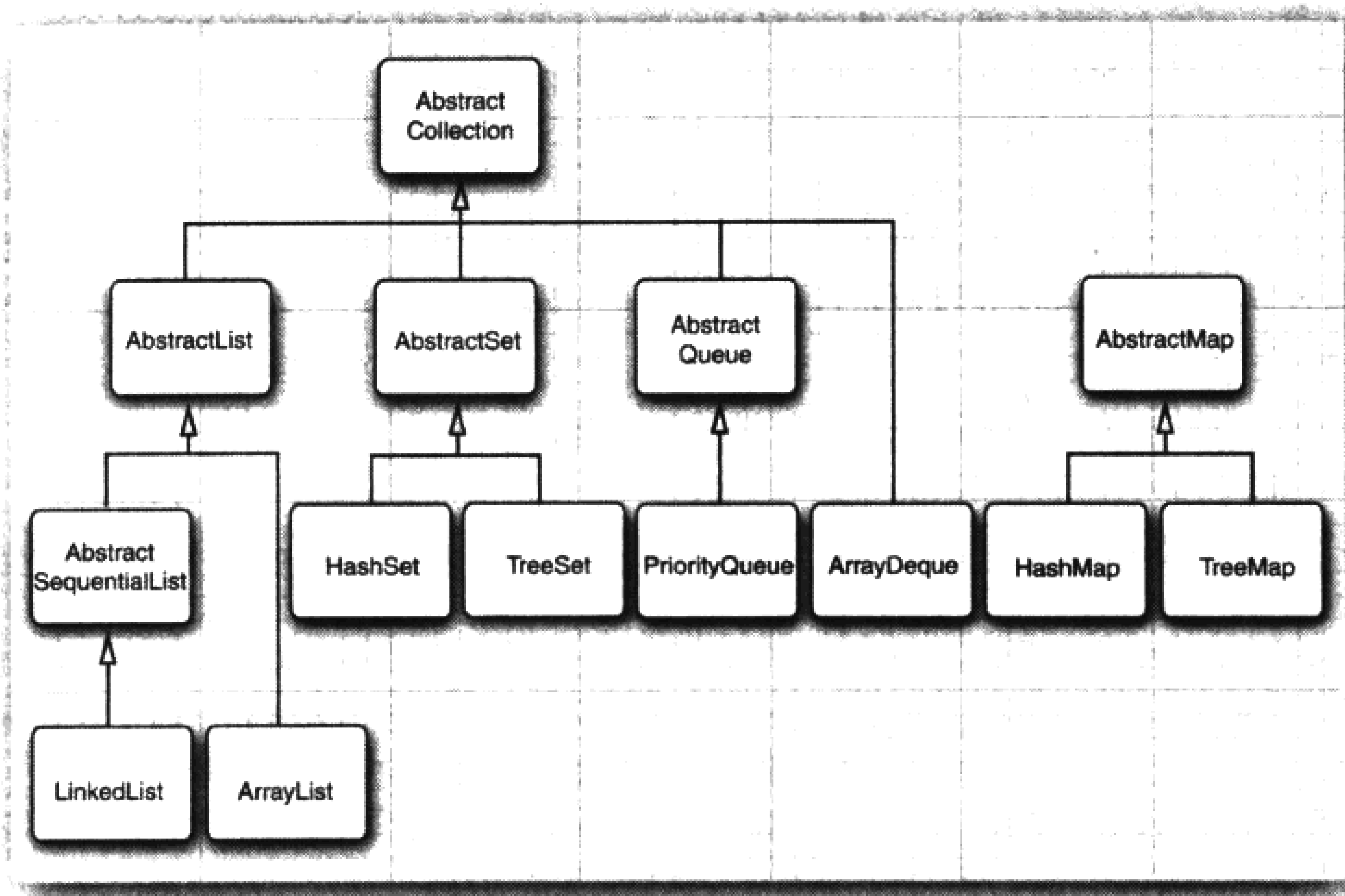


图13-11 集合框架中的类

最后，还有许多Java第一版“遗留”下来的容器类，在集合框架出现就有了，它们是：

```
Vector
Stack
Hashtable
Properties
```

这些类已经被集成到集合框架中，见图13-12。本章稍后将会讨论这些类。

13.3.1 视图与包装器

看一下图13-10和图13-11可能会感觉：用如此多的接口和抽象类来实现数量并不多的具体集合类似乎没有太大必要。然而，这两张图并没有展示出全部的情况。通过使用视图（views）可以获得其他的实现了集合接口和映射表接口的对象。映射表类的keySet方法就是一个这样的示例。初看起来，好像这个方法创建了一个新集，并将映射表中的所有键都填进去，然后返回这个集。但是，情况并非如此。取而代之的是：keySet方法返回一个实现Set接口的类对象，这个类的方法对原映射表进行操作。这种集合称为视图。

视图技术在集框架中有许多非常有用的应用。下面将讨论这些应用。

1. 轻量级集包装器

Arrays类的静态方法asList将返回一个包装了普通Java数组的List包装器。这个方法可以将数组传递给一个期望得到列表或集合变元的方法。例如：

```
Card[] cardDeck = new Card[52];
. . .
```

```
List<Card> cardList = Arrays.asList(cardDeck);
```

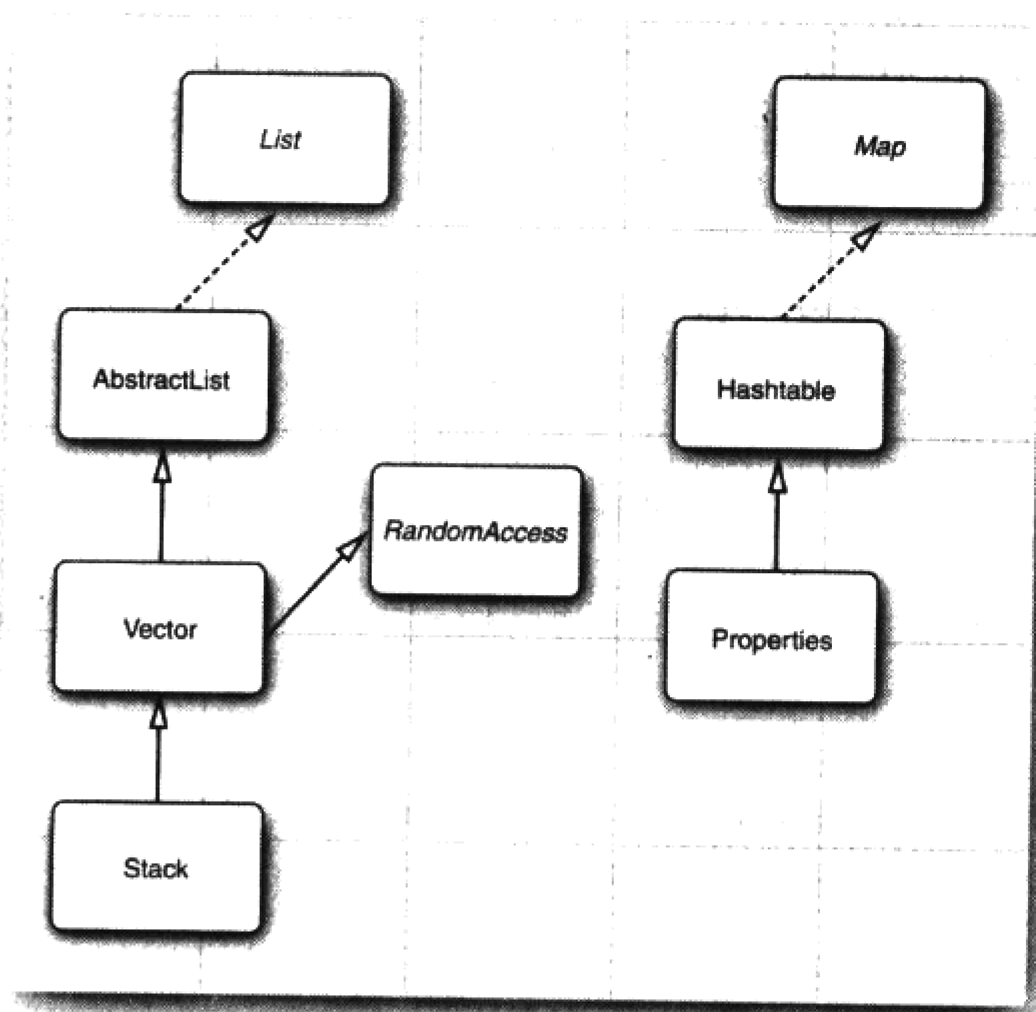


图13-12　集合框架中的遗留类

返回的对象不是ArrayList。它是一个视图对象，带有访问底层数组的get和set方法。改变数组大小的所有方法（例如，与迭代器相关的add和remove方法）都会抛出一个UnsupportedOperationException异常。

从Java SE 5.0开始，asList方法声明为一个具有可变数量参数的方法。除了可以传递一个数组之外，还可以将各个元素直接传递给这个方法。例如：

```
List<String> names = Arrays.asList("Amy", "Bob", "Carl");
```

这个方法调用

```
Collections.nCopies(n, anObject)
```

将返回一个实现了List接口的不可修改的对象，并给人一种包含*n*个元素，每个元素都像是一个anObject的错觉。

例如，下面的调用将创建一个包含100个字符串的List，每个串都被设置为“DEFAULT”：

```
List<String> settings = Collections.nCopies(100, "DEFAULT");
```

由于字符串对象只存储了一次，所以付出的存储代价很小。这是视图技术的一种巧妙的应用。

✔ **注释**：Collections类包含很多实用方法，这些方法的参数和返回值都是集合。不要将它与Collection接口混淆起来。

如果调用下列方法

```
Collections.singleton(anObject)
```

则将返回一个视图对象。这个对象实现了Set接口（与产生List的ncopies方法不同）。返回的对象实现了一个不可修改的单元素集，而不需要付出建立数据结构的开销。singletonList方法与singletonMap方法类似。

2. 子范围

可以为很多集合建立子范围（subrange）视图。例如，假设有一个列表staff，想从中取出第10个~第19个元素。可以使用subList方法来获得一个列表的子范围视图。

```
List group2 = staff.subList(10, 20);
```

第一个索引包含在内，第二个索引则不包含在内。这与String类的substring操作中的参数情况相同。

可以将任何操作应用于子范围，并且能够自动地反映整个列表的情况。例如，可以删除整个子范围：

```
group2.clear(); // staff reduction
```

现在，元素自动地从staff列表中清除了，并且group2为空。

对于有序集和映射表，可以使用排序顺序而不是元素位置建立子范围。SortedSet接口声明了3个方法：

```
SortedSet<E> subSet(E from, E to)
SortedSet<E> headSet(E to)
SortedSet<E> tailSet(E from)
```

这些方法将返回大于等于from且小于to的所有元素子集。有序映射表也有类似的方法：

```
SortedMap<K, V> subMap(K from, K to)
SortedMap<K, V> headMap(K to)
SortedMap<K, V> tailMap(K from)
```

返回映射表视图，该映射表包含键落在指定范围内的所有元素。

Java SE 6引入的NavigableSet接口赋予子范围操作更多的控制能力。可以指定是否包括边界：

```
NavigableSet<E> subSet(E from, boolean fromInclusive, E to, boolean toInclusive)
NavigableSet<E> headSet(E to, boolean toInclusive)
NavigableSet<E> tailSet(E from, boolean fromInclusive)
```

3. 不可修改的视图

Collections还有几个方法，用于产生集合的不可修改视图（unmodifiable views）。这些视图对现有集合增加了一个运行时的检查。如果发现试图对集合进行修改，就抛出一个异常，同时这个集合将保持未修改的状态。

可以使用下面6种方法获得不可修改视图：

```
Collections.unmodifiableCollection
Collections.unmodifiableList
Collections.unmodifiableSet
Collections.unmodifiableSortedSet
Collections.unmodifiableMap
```

```
Collections.unmodifiableSortedMap
```

每个方法都定义于一个接口。例如，Collections.unmodifiableList与ArrayList、LinkedList或者任何实现了List接口的其他类一起协同工作。

例如，假设想要查看某部分代码，但又不触及某个集合的内容，就可以进行下列操作：

```
List<String> staff = new LinkedList<String>();
. . .
lookAt(new Collections.unmodifiableList(staff));
```

Collections.unmodifiableList方法将返回一个实现List接口的类对象。其访问器方法将从staff集合中获取值。当然，lookAt方法可以调用List接口中的所有方法，而不只是访问器。但是所有的更改器方法（例如，add）已经被重新定义为抛出一个UnsupportedOperationException异常，而不是将调用传递给底层集合。

不可修改视图并不是集合本身不可修改。仍然可以通过集合的原始引用（在这里是staff）对集合进行修改。并且仍然可以让集合的元素调用更改器方法。

由于视图只是包装了接口而不是实际的集合对象，所以只能访问接口中定义的方法。例如，LinkedList类有一些非常方便的方法，addFirst和addLast，它们都不是List接口的方法，不能通过不可修改视图进行访问。

警告：unmodifiableCollection方法（与本节稍后讨论的synchronizedCollection和checkedCollection方法一样）将返回一个集合，它的equals方法不调用底层集合的equals方法。相反，它继承了Object类的equals方法，这个方法只是检测两个对象是否是同一个对象。如果将集或列表转换成集合，就再也无法检测其内容是否相同了。视图就是以这种方式运行的，因为内容是否相等的检测在分层结构的这一层上没有定义妥当。视图将以同样的方式处理hashCode方法。

然而，unmodifiableSet类和unmodifiableList类却使用底层集合的equals方法和hashCode方法。

4. 同步视图

如果由多个线程访问集合，就必须确保集不会被意外地破坏。例如，如果一个线程试图将元素添加到散列表中，同时另一个线程正在对散列表进行再散列，其结果将是灾难性的。

类库的设计者使用视图机制来确保常规集合的线程安全，而不是实现线程安全的集合类。例如，Collections类的静态synchronizedMap方法可以将任何一个映射表转换成具有同步访问方法的Map：

```
Map<String, Employee> map = Collections.synchronizedMap(new HashMap<String, Employee>());
```

现在，就可以由多线程访问map对象了。像get和put这类方法都是串行操作的，即在另一个线程调用另一个方法之前，刚才的方法调用必须彻底完成。第14章将会详细地讨论数据结构的同步访问。

5. 被检验视图

Java SE 5.0 增加了一组“被检验”视图，用来对泛型类型发生问题时提供调试支持。如同

第12章中所述，实际上将错误类型的元素私自带到泛型集合中的问题极有可能发生。例如：

```
ArrayList<String> strings = new ArrayList<String>();
ArrayList rawList = strings; // get warning only, not an error, for compatibility with legacy code
rawList.add(new Date()); // now strings contains a Date object!
```

这个错误的add命令在运行时检测不到。相反，只有在稍后的另一部分代码中调用get方法，并将结果转化为String时，这个类才会抛出异常。被检验视图可以探测到这类问题。下面定义了一个安全列表：

```
List<String> safeStrings = Collections.checkedList(strings, String.class);
```

视图的add方法将检测插入的对象是否属于给定的类。如果不属于给定的类，就立即抛出一个ClassCastException。这样做的好处是错误可以在正确的位置得以报告：

```
ArrayList rawList = safeStrings;
rawList.add(new Date()); // Checked list throws a ClassCastException
```

警告：被检测视图受限于虚拟机可以运行的运行时检查。例如，对于Array List <Pair<String>>，由于虚拟机有一个单独的“原始”Pair类，所以，无法阻止插入 Pair <Date>。

6. 关于可选操作的说明

通常，视图有一些局限性，即可能只可以读、无法改变大小、只支持删除而不支持插入，这些与映射表的键视图情况相同。如果试图进行不恰当的操作，受限制的视图就会抛出一个UnsupportedOperationException。

在集合和迭代器接口的API文档中，许多方法描述为“可选操作”。这看起来与接口的概念有所抵触。毕竟，接口的设计目的难道不是负责给出一个类必须实现的方法吗？确实，从理论的角度看，在这里给出的方法很难令人满意。一个更好的解决方案是为每个只读视图和不能改变集合大小的视图建立各自独立的两个接口。不过，这将会使接口的数量成倍增长，这让类库设计者无法接受。

是否应该将“可选”方法这一技术扩展到用户的设计中呢？我们认为不应该。尽管集合被频繁地使用，其实现代码的风格也未必适用于其他问题领域。集合类库的设计者必须解决一组特别严格且又相互冲突的需求。用户希望类库应该易于学习、使用方便，彻底泛型化，面向通用性，同时又与手写算法一样高效。要同时达到所有目标的要求，或者尽量兼顾所有目标完全是不可能的。但是，在自己的编程问题中，很少遇到这样极端的局限性。应该能够找到一种不必依靠极端衡量“可选的”接口操作来解决这类问题的方案。

API java.util.Collections 1.2

- static <E> Collection unmodifiableCollection(Collection<E> c)
- static <E> List unmodifiableList(List<E> c)
- static <E> Set unmodifiableSet(Set<E> c)
- static <E> SortedSet unmodifiableSortedSet(SortedSet<E> c)
- static <K, V> Map unmodifiableMap(Map<K, V> c)

• static <K, V> SortedMap unmodifiableSortedMap(SortedMap<K, V> c)

构造一个集合视图，其更改器方法将抛出一个UnsupportedOperationException。

• static <E> Collection<E> synchronizedCollection(Collection<E> c)
• static <E> List synchronizedList(List<E> c)
• static <E> Set synchronizedSet(Set<E> c)
• static <E> SortedSet synchronizedSortedSet(SortedSet<E> c)
• static <K, V> Map<K, V> synchronizedMap(Map<K, V> c)
• static <K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> c)

构造一个集合视图，其方法都是同步的。

• static <E> Collection checkedCollection(Collection<E> c, Class<E> elementType)
• static <E> List checkedList(List<E> c, Class<E> elementType)
• static <E> Set checkedSet(Set<E> c, Class<E> elementType)
• static <E> SortedSet checkedSortedSet(SortedSet<E> c, Class<E> elementType)
• static <K, V> Map checkedMap(Map<K, V> c, Class<K> keyType, Class<V> valueType)
• static <K, V> SortedMap checkedSortedMap(SortedMap<K, V> c, Class<K> keyType, Class<V> valueType)

构造一个集合视图。如果插入一个错误类型的元素，将抛出一个ClassCastException。

• static <E> List<E> nCopies(int n, E value)
• static <E> Set<E> singleton(E value)

构造一个对象视图，它既可以作为一个拥有*n*个相同元素的不可修改列表，又可以作为一个拥有单个元素的集。

API java.util.Arrays 1.2

• static <E> List<E> asList(E... array)

返回一个数组元素的列表视图。这个数组是可修改的，但其大小不可变。

API java.util.List<E> 1.2

• List<E> subList(int firstIncluded, int firstExcluded)

返回给定位置范围内的所有元素的列表视图。

API java.util.SortedSet<E> 1.2

• SortedSet<E> subSet(E firstIncluded, E firstExcluded)
• SortedSet<E> headSet(E firstExcluded)
• SortedSet<E> tailSet(E firstIncluded)

返回给定范围内的元素视图。

java.util.NavigableSet<E> 6

- NavigableSet<E> subSet(E from, boolean fromIncluded, E to, boolean toIncluded)
- NavigableSet<E> headSet(E to, boolean toIncluded)
- NavigableSet<E> tailSet(E from, boolean fromIncluded)

  返回给定范围内的元素视图。boolean 标志决定视图是否包含边界。

java.util.SortedMap<K, V> 1.2

- SortedMap<K, V> subMap(K firstIncluded, K firstExcluded)
- SortedMap<K, V> headMap(K firstExcluded)
- SortedMap<K, V> tailMap(K firstIncluded)

  返回在给定范围内的键条目的映射表视图。

java.util.NavigableMap<K, V> 6

- NavigableMap<K, V> subMap(K from, boolean fromIncluded, K to, boolean toIncluded)
- NavigableMap<K, V> headMap(K from, boolean fromIncluded)
- NavigableMap<K, V> tailMap(K to, boolean toIncluded)

  返回在给定范围内的键条目的映射表视图。boolean标志决定视图是否包含边界。

13.3.2 批操作

到现在为止，列举的绝大多数示例都采用迭代器遍历集合，一次遍历一个元素。然而，可以使用类库中的*批操作*（bulk operations）避免频繁地使用迭代器。

假设希望找出两个集的*交*（intersection），即两个集中共有的元素。首先，要建立一个新集，用于存放结果。

```
Set<String> result = new HashSet<String>(a);
```

这里利用了这样一个事实：每一个集合有一个构造器，其参数是保存初始值的另一个集合。

接着，调用retainAll方法：

```
result.retainAll(b);
```

result中保存了既在a中出现，也在b中出现的元素。这时已经构成了交集，而且没有使用循环。

可以将这个思路向前推进一步，将批操作应用于视图。例如，假如有一个映射表，将员工的ID映射为员工对象，并且建立了一个将要结束聘用期的所有员工的ID集。

```
Map<String, Employee> staffMap = . . .;
Set<String> terminatedIDs = . . .;
```

直接建立一个键集，并删除终止聘用关系的所有员工的ID即可。

```
staffMap.keySet().removeAll(terminatedIDs);
```

由于键集是映射表的一个视图，所以，键与对应的员工名将会从映射表中自动地删除。

通过使用一个子范围视图，可以将批操作限制于子列表和子集的操作上。例如，假设希望将一个列表的前10个元素添加到另一个容器中，可以建立一个子列表用于选择前10个元素：

```
relocated.addAll(staff.subList(0, 10));
```

这个子范围也可以成为更改操作的对象。

```
staff.subList(0, 10).clear();
```

13.3.3 集合与数组之间的转换

由于Java平台API中的大部分内容都是在集合框架创建之前设计的，所以，有时候需要在传统的数组与现代的集合之前进行转换。

如果有一个数组需要转换为集合。Arrays.asList的包装器就可以实现这个目的。例如：

```
String[] values = . . .;
HashSet<String> staff = new HashSet<String>(Arrays.asList(values));
```

反过来，将集合转换为数组就有点难了。当然，可以使用toArray方法：

```
Object[] values = staff.toArray();
```

但是，这样做的结果是产生一个对象数组。即使知道集合中包含一个特定类型的对象，也不能使用类型转换：

```
String[] values = (String[]) staff.toArray(); // Error!
```

由toArray方法返回的数组是一个Object[]数组，无法改变其类型。相反，必须使用另外一种toArray方法，并将其设计为所希望的元素类型且长度为0的数组。随后返回的数组将与所创建的数组一样：

```
String[] values = staff.toArray(new String[0]);
```

如果愿意的话，可以构造一个指定大小的数组：

```
staff.toArray(new String[staff.size()]);
```

在这种情况下，没有创建任何一个新数组。

✔ **注释**：为什么不直接将一个Class对象（例如，String.class）传递给toArray方法。其原因是这个方法具有“双重职责”，不仅要填充已有的数组（如果足够长），还要创建一个新数组。

13.4 算法

泛型集合接口有一个很大的优点，即算法只需要实现一次。例如，考虑一下计算集合中最大元素这样一个简单的算法。使用传统方式，程序设计人员可能会用循环实现这个算法。下面就是找出数组中最大元素的代码。

```
if (a.length == 0) throw new NoSuchElementException();
T largest = a[0];
for (int i = 1; i < a.length; i++)
   if (largest.compareTo(a[i]) < 0)
      largest = a[i];
```

当然，为找出数组列表中的最大元素所编写的代码会与此有微小的差别。

```
if (v.size() == 0) throw new NoSuchElementException();
T largest = v.get(0);
for (int i = 1; i < v.size(); i++)
   if (largest.compareTo(v.get(i)) < 0)
      largest = v.get(i);
```

链表应该怎么做呢？对于链表来说，无法实施高效的随机访问，但却可以使用迭代器。

```
if (l.isEmpty()) throw new NoSuchElementException();
Iterator<T> iter = l.iterator();
T largest = iter.next();
while (iter.hasNext())
{
   T next = iter.next();
   if (largest.compareTo(next) < 0)
      largest = next;
}
```

编写这些循环代码有些乏味，并且也很容易出错。是否存在严重错误吗？对于空容器循环能正常工作吗？对于只含有一个元素的容器又会发生什么情况呢？我们不希望每次都测试和调试这些代码，也不想实现下面这一系列的方法：

```
static <T extends Comparable> T max(T[] a)
static <T extends Comparable> T max(ArrayList<T> v)
static <T extends Comparable> T max(LinkedList<T> l)
```

这正是集合接口的用武之地。仔细考虑一下，为了高效地使用这个算法所需要的最小集合接口。采用get和set方法进行随机访问要比直接迭代层次高。在计算链表中最大元素的过程中已经看到，这项任务并不需要进行随机访问。直接用迭代器遍历每个元素就可以计算最大元素。因此，可以将max方法实现为能够接收任何实现了Collection接口的对象。

```
public static <T extends Comparable> T max(Collection<T> c)
{
   if (c.isEmpty()) throw new NoSuchElementException();
   Iterator<T> iter = c.iterator();
   T largest = iter.next();
   while (iter.hasNext())
   {
      T next = iter.next();
      if (largest.compareTo(next) < 0)
         largest = next;
   }
   return largest;
}
```

现在就可以使用一个方法计算链表、数组列表或数组中最大元素了。

这是一个非常重要的概念。事实上，标准的C++类库已经有几十种非常有用的算法，每个算法都是在泛型集合上操作的。Java类库中的算法没有如此丰富，但是，也包含了基本的排序、二分查找等实用算法。

13.4.1 排序与混排

计算机行业的前辈们有时会回忆起他们当年不得不使用穿孔卡片以及手工地编写排序算法的情形。当然，如今排序算法已经成为大多数编程语言标准库中的一个组成部分，Java程序设计语言也不例外。

Collections类中的sort方法可以对实现了List接口的集合进行排序。

```
List<String> staff = new LinkedList<String>();
// fill collection . . .;
Collections.sort(staff);
```

这个方法假定列表元素实现了Comparable接口。如果想采用其他方式对列表进行排序，可以将Comparator对象作为第二个参数传递给sort方法。（已经在前面的章节中介绍过比较器），下面的代码说明了对列表中各项进行排序的基本方法：

```
Comparator<Item> itemComparator = new
   Comparator<Item>()
   {
      public int compare(Item a, Item b)
      {
         return a.partNumber - b.partNumber;
      }
   });
Collections.sort(items, itemComparator);
```

如果想按照降序对列表进行排序，可以使用一种非常方便的静态方法Collections.reverseOrder()。这个方法将返回一个比较器，比较器则返回b.compareTo(a)。例如，

```
Collections.sort(staff, Collections.reverseOrder())
```

这个方法将根据元素类型的compareTo方法给定排序顺序，按照逆序对列表staff进行排序。同样，

```
Collections.sort(items, Collections.reverseOrder(itemComparator))
```

将逆置itemComparator的次序。

人们可能会对sort方法所采用的排序手段感到好奇。通常，在翻阅有关算法书籍中的排序算法时，会发觉介绍的都是有关数组的排序算法，而且使用的是随机访问方式。但是，对列表进行随机访问的效率很低。实际上，可以使用归并排序对列表进行高效的排序（例如，可以参看Addison-Wesley出版社1998年出版的Robert Sedgewick编写的《Algorithms in C++》第366～369页）。然而，Java程序设计语言并不是这样实现的。它直接将所有元素转入一个数组，并使用一种归并排序的变体对数组进行排序，然后，再将排序后的序列复制回列表。

集合类库中使用的归并排序算法比快速排序要慢一些，快速排序是通用排序算法的传统选择。但是，归并排序有一个主要的优点：*稳定*，即不需要交换相同的元素。为什么要关注相同元素的顺序呢？下面是一种常见的情况。假设有一个已经按照姓名排列的员工列表。现在，要按照工资再进行排序。如果两个雇员的工资相等发生什么情况呢？如果采用稳定的排序算法，将会保留按名字排列的顺序。换句话说，排序的结果将会产生这样一个列表，首先按照工资排序，工资相同者再按照姓名排序。

因为集合不需要实现所有的“可选”方法，因此，所有接受集合参数的方法必须描述什么时候可以安全地将集合传递给算法。例如，显然不能将unmodifiableList列表传递给排序算法。可以传递什么类型的列表呢？根据文档说明，列表必须是可修改的，但不必是可以改变大小的。

下面是有关的术语定义：

- 如果列表支持set方法，则是*可修改的*。
- 如果列表支持add和remove方法，则是*可改变大小的*。

Collections类有一个算法shuffle，其功能与排序刚好相反，即随机地混排列表中元素的顺序。例如：

```
ArrayList<Card> cards = . . .;
Collections.shuffle(cards);
```

如果提供的列表没有实现RandomAccess接口，shuffle方法将元素复制到数组中，然后打乱数组元素的顺序，最后再将打乱顺序后的元素复制回列表。

例13-6的程序用1～49之间的49个Integer对象填充数组。然后，随机地打乱列表，并从打乱后的列表中选前6个值。最后再将选择的数值进行排序和打印。

例13-6 ShuffleTest.java

```
import java.util.*;

/**
 * This program demonstrates the random shuffle and sort algorithms.
 * @version 1.10 2004-08-02
 * @author Cay Horstmann
 */
public class ShuffleTest
{
   public static void main(String[] args)
   {
      List<Integer> numbers = new ArrayList<Integer>();
      for (int i = 1; i <= 49; i++)
         numbers.add(i);
      Collections.shuffle(numbers);
      List<Integer> winningCombination = numbers.subList(0, 6);
      Collections.sort(winningCombination);
      System.out.println(winningCombination);
   }
}
```

API java.util.Collections 1.2

- `static <T extends Comparable<? super T>> void sort(List<T> elements)`
- `static <T> void sort(List<T> elements, Comparator<? super T> c)`

  使用稳定的排序算法，对列表中的元素进行排序。这个算法的时间复杂度是O(n log n)，其中n为列表的长度。

- `static void shuffle(List<?> elements)`
- `static void shuffle(List<?> elements, Random r)`

随机地打乱列表中的元素。这个算法的时间复杂度是 $O(n\ a(n))$，n是列表的长度，$a\ (n)$ 是访问元素的平均时间。

- `static <T> Comparator<T> reverseOrder()`

  返回一个比较器，它用与Comparable接口的compareTo方法规定的顺序的逆序对元素进行排序。

- `static <T> Comparator<T> reverseOrder(Comparator<T> comp)`

  返回一个比较器，它用与comp给定的顺序的逆序对元素进行排序。

13.4.2 二分查找

要想在数组中查找一个对象，通常要依次访问数组中的每个元素，直到找到匹配的元素为止。然而，如果数组是有序的，就可以直接查看位于数组中间的元素，看一看是否大于要查找的元素。如果是，用同样的方法在数组的前半部分继续查找；否则，用同样的方法在数组的后半部分继续查找。这样就可以将查找范围缩减一半。一直用这种方式查找下去。例如，如果数组中有1024个元素，可以在10次比较后定位所匹配的元素（或者可以确认在数组中不存在这样的元素），而使用线性查找，如果元素存在，平均需要512次比较；如果元素不存在，需要1024次比较才可以确认。

Collections类的binarySearch方法实现了这个算法。注意，集合必须是排好序的，否则算法将返回错误的答案。要想查找某个元素，必须提供集合（这个集合要实现List接口，下面还要更加详细地介绍这个问题）以及要查找的元素。如果集合没有采用Comparable接口的compareTo方法进行排序，就还要提供一个比较器对象。

```
i = Collections.binarySearch(c, element);
i = Collections.binarySearch(c, element, comparator);
```

如果binarySearch方法返回的数值大于等于0，则表示匹配对象的索引。也就是说，c.get(i)等于在这个比较顺序下的element。如果返回负值，则表示没有匹配的元素。但是，可以利用返回值计算应该将element插入到集合的哪个位置，以保持集合的有序性。插入的位置是

```
insertionPoint = -i - 1;
```

这并不是简单的－i，因为0值是不确定的。也就是说，下面这个操作：

```
if (i < 0)
   c.add(-i - 1, element);
```

将把元素插入到正确的位置上。

只有采用随机访问，二分查找才有意义。如果必须利用迭代方式一次次地遍历链表的一半元素来找到中间位置的元素，二分查找就完全失去了优势。因此，如果为binarySearch算法提供一个链表，它将自动地变为线性查找。

✔ **注释**：在Java SE 1.3中，没有为有序集合提供专门的接口，以进行高效地随机访问，而binarySearch方法使用的是一种拙劣的策略，即检查列表参数是否扩展了Abstract Sequential List类。这个问题在Java SE 1.4中得到了解决。现在binarySearch方法检查列表参数是否实现了RandomAccess接口。如果实现了这个接口，这个方法将采用二分查找；否则，将采用

线性查找。

API java.util.Collections 1.2

- static <T extends Comparable<? super T>> int binarySearch(List<T> elements, T key)
- static <T> int binarySearch(List<T> elements, T key, Comparator<? super T> c)

  从有序列表中搜索一个键，如果元素扩展了AbstractSequentialList类，则采用线性查找，否则将采用二分查找。这个方法的时间复杂度为O (a (n) log n)，n是列表的长度，a (n)是访问一个元素的平均时间。这个方法将返回这个键在列表中的索引，如果在列表中不存在这个键将返回负值i。在这种情况下，应该将这个键插入到列表索引－i－1的位置上，以保持列表的有序性。

13.4.3 简单算法

在Collections类中包含了几个简单且很有用的算法。前面介绍的查找集合中最大元素的示例就在其中。另外还包括：将一个列表中的元素复制到另外一个列表中；用一个常量值填充容器；逆置一个列表的元素顺序。为什么会在标准库中提供这些简单算法呢？大多数程序员肯定可以很容易地采用循环实现这些算法。我们之所以喜欢这些算法是因为：它们可以让程序员阅读算法变成一件轻松的事情。当阅读由别人实现的循环时，必须要揣摩编程者的意图。而在看到诸如Collections.max这样的方法调用时，一定会立刻明白其用途。

下面的API注释描述了Collections类的一些简单算法。

API java.util.Collections 1.2

- static <T extends Comparable<? super T>> T min(Collection<T> elements)
- static <T extends Comparable<? super T>> T max(Collection<T> elements)
- static <T> min(Collection<T> elements, Comparator<? super T> c)
- static <T> max(Collection<T> elements, Comparator<? super T> c)

  返回集合中最小的或最大的元素（为清楚起见，参数的边界被简化了）。
- static <T> void copy(List<? super T> to, List<T> from)

  将原列表中的所有元素复制到目标列表的相应位置上。目标列表的长度至少与原列表一样。
- static <T> void fill(List<? super T> l, T value)

  将列表中所有位置设置为相同的值。
- static <T> boolean addAll(Collection<? super T> c, T... values) 5.0

  将所有的值添加到集合中。如果集合改变了，则返回true。
- static <T> boolean replaceAll(List<T> l, T oldValue, T newValue) 1.4

  用newValue取代所有值为oldValue的元素。
- static int indexOfSubList(List<?> l, List<?> s) 1.4
- static int lastIndexOfSubList(List<?> l, List<?> s) 1.4

返回l中第一个或最后一个等于s子列表的索引。如果l中不存在等于s的子列表，则返回－1。例如，l为[s, t, a, r]，s为[t, a, r]，两个方法都将返回索引1。

- `static void swap(List<?> l, int i, int j) 1.4`

  交换给定偏移量的两个元素。

- `static void reverse(List<?> l)`

  逆置列表中元素的顺序。例如，逆置列表 [t, a, r] 后将得到列表 [r, a, t]。这个方法的时间复杂度为O (n)，n为列表的长度。

- `static void rotate(List<?> l, int d) 1.4`

  旋转列表中的元素，将索引i的条目移动到位置(i + d) % l.size()。例如，将列表[t, a, r] 旋转移2个位置后得到[a, r, t]。这个方法的时间复杂度为O(n)，n为列表的长度。

- `static int frequency(Collection<?> c, Object o) 5.0`

  返回c中与对象o相同的元素个数。

- `boolean disjoint(Collection<?> c1, Collection<?> c2) 5.0`

  如果两个集合没有共同的元素，则返回true。

13.4.4 编写自己的算法

如果编写自己的算法（实际上，是以集合作为参数的任何方法），应该尽可能地使用接口，而不要使用具体的实现。例如，假设想用一组菜单项填充JMenu。传统上，这种方法可能会按照下列方式实现：

```
void fillMenu(JMenu menu, ArrayList<JMenuItem> items)
{
   for (JMenuItem item : items)
      menu.addItem(item);
}
```

但是，这样会限制方法的调用程序，即调用程序必须在ArrayList中提供选项。如果这些选项需要放在另一个容器中，首先必须对它们重新包装，因此，最好接受一个更加通用的集合。

什么是完成这项工作的最通用的集合接口？在这里，只需要访问所有的元素，这是Collection接口的基本功能。下面代码说明了如何重新编写fillMenu方法使之接受任意类型的集合。

```
void fillMenu(JMenu menu, Collection<JMenuItem> items)
{
   for (JMenuItem item : items)
      menu.addItem(item);
}
```

现在，任何人都可以用ArrayList或LinkedList，甚至用Arrays.asList包装器包装的数组调用这个方法。

注释：既然将集合接口作为方法参数是个很好的想法，为什么Java类库不更多地这样做呢？例如，JComboBox有两个构造器：

```
JComboBox(Object[] items)
JComboBox(Vector<?> items)
```

之所以没有这样做，原因很简单：时间问题。Swing类库是在集合类库之前创建的。

如果编写了一个返回集合的方法，可能还想要一个返回接口，而不是返回类的方法，因为这样做可以在日后改变想法，并用另一个集合重新实现这个方法。

例如，编写一个返回所有菜单项的方法getAllItems。

```
List<MenuItem> getAllItems(JMenu menu)
{
   ArrayList<MenuItem> items = new ArrayList<MenuItem>()
   for (int i = 0; i < menu.getItemCount(); i++)
      items.add(menu.getItem(i));
   return items;
}
```

日后，可以做出这样的决定：不复制所有的菜单项，而仅仅提供这些菜单项的视图。要做到这一点，只需要返回AbstractList的匿名子类。

```
List<MenuItem> getAllItems(final JMenu menu)
{
   return new
      AbstractList<MenuItem>()
      {
         public MenuItem get(int i)
         {
            return item.getItem(i);
         }
         public int size()
         {
            return item.getItemCount();
         }
      };
}
```

当然，这是一项高级技术。如果使用它，就应该将它支持的那些“可选”操作准确地记录在文档中。在这种情况下，必须提醒调用者返回的对象是一个不可修改的列表。

13.5 遗留的集合

本节将讨论Java程序设计语言自问世以来就存在的集合类：Hashtable类和非常有用的子类Properties、Vector的子类Stack以及BitSet类。

13.5.1 Hashtable类

Hashtable类与HashMap类的作用一样，实际上，它们拥有相同的接口。与Vector类的方法一样。Hashtable的方法也是同步的。如果对同步性或与遗留代码的兼容性没有任何要求，就应该使用HashMap。

注释：这个类的名字是Hashtable，带有一个小写的t。在Windows操作系统下，如果使用HashTable会看到一个很奇怪的错误信息，这是因为Windows文件系统对大小写不敏感，而Java编译器却对大小写敏感。

13.5.2 枚举

遗留集合使用Enumeration接口对元素序列进行遍历。Enumeration接口有两个方法，即hasMoreElements和nextElement。这两个方法与Iterator接口的hasNext方法和next方法十分类似。

例如，Hashtable类的elements方法将产生一个用于描述表中各个枚举值的对象：

```
Enumeration<Employee> e = staff.elements();
while (e.hasMoreElements())
{
   Employee e = e.nextElement();
   . . .
}
```

有时还会遇到遗留的方法，其参数是枚举类型的。静态方法 Collections.enumeration将产生一个枚举对象，枚举集合中的元素。例如：

```
List<InputStream> streams = . . .;
SequenceInputStream in = new SequenceInputStream(Collections.enumeration(streams));
  // the SequenceInputStream constructor expects an enumeration
```

注释：在C++中，用迭代器作为参数十分普遍。幸好，在Java的编程平台中，只有极少的程序员沿用这种习惯。传递集合要比传递迭代器更为明智。集合对象的用途更大。当接受方如果需要时，总是可以从集合中获得迭代器，而且，还可以随时地使用集合的所有方法。不过，可能会在某些遗留代码中发现枚举接口，因为这是在Java SE 1.2的集合框架出现之前，它们是泛型集合惟一可以使用的机制。

API `java.util.Enumeration<E>` 1.0

- `boolean hasMoreElements()`

  如果还有更多的元素可以查看，则返回true。
- `E nextElement()`

  返回被检测的下一个元素。如果hasMoreElements()返回false，则不要调用这个方法。

API `java.util.Hashtable<K, V>` 1.0

- `Enumeration<K> keys()`

  返回一个遍历散列表中键的枚举对象。
- `Enumeration<V> elements()`

  返回一个遍历散列表中元素的枚举对象。

API `java.util.Vector<E>` 1.0

- `Enumeration<E> elements()`

  返回遍历向量中元素的枚举对象。

13.5.3 属性映射表

属性映射表（property map）是一个类型非常特殊的映射表结构。它有下面3个特性：

- 键与值都是字符串。
- 表可以保存到一个文件中，也可以从文件中加载。
- 使用一个默认的辅助表。

实现属性映射表的Java平台类称为Properties。

属性映射表通常用于程序的特殊配置选项，参见第10章。

API `java.util.Properties` 1.0

- `Properties()`

  创建一个空的属性映射表。
- `Properties(Properties defaults)`

  创建一个带有一组默认值的空的属性映射表。
- `String getProperty(String key)`

  获得属性的对应关系；返回与键对应的字符串。如果在映射表中不存在，返回默认表中与这个键对应的字符串。
- `String getProperty(String key, String defaultValue)`

  获得在键没有找到时具有的默认值属性；它将返回与键对应的字符串，如果在映射表中不存在，就返回默认的字符串。
- `void load(InputStream in)`

  从InputStream加载属性映射表。
- `void store(OutputStream out, String commentString)`

  把属性映射表存储到OutputStream。

13.5.4 栈

从1.0版开始，标准类库中就包含了Stack类，其中有大家熟悉的push方法和pop方法。但是，Stack类扩展为Vector类，从理论角度看，Vector类并不太令人满意，它可以让栈使用不属于栈操作的insert和remove方法，即可以在任何地方进行插入或删除操作，而不仅仅是在栈顶。

API `java.util.Stack<E>` 1.0

- `E push(E item)`

  将item压入栈并返回item。
- `E pop()`

  弹出并返回栈顶的item。如果栈为空，请不要调用这个方法。
- `E peek()`

  返回栈顶元素，但不弹出。如果栈为空，请不要调用这个方法。

13.5.5 位集

Java平台的BitSet类用于存放一个位序列（它不是数学上的集，称为位向量或位数组更为合适）。如果需要高效地存储位序列（例如，标志）就可以使用位集。由于位集将位包装在字节里，所以，使用位集要比使用Boolean对象的ArrayList更加高效。

BitSet类提供了一个便于读取、设置或清除各个位的接口。使用这个接口可以避免屏蔽和其他麻烦的位操作。如果将这些位存储在int或long变量中就必须进行这些繁琐的操作。

例如，对于一个名为bucketOfBits的BitSet，

```
bucketOfBits.get(i)
```

如果第i位处于“开”状态，就返回true；否则返回false。同样地，

```
bucketOfBits.set(i)
```

将第i位置为“开”状态。最后，

```
bucketOfBits.clear(i)
```

将第i位置为“关”状态。

C++注释：C++位集模板与Java平台的BitSet功能一样。

API java.util.BitSet 1.0

- `BitSet(int initialCapacity)`
  创建一个位集。
- `int length()`
  返回位集的“逻辑长度”，即1加上位集的最高设置位的索引。
- `boolean get(int bit)`
  获得一个位。
- `void set(int bit)`
  设置一个位。
- `void clear(int bit)`
  清除一个位。
- `void and(BitSet set)`
  这个位集与另一个位集进行逻辑“AND”。
- `void or(BitSet set)`
  这个位集与另一个位集进行逻辑“OR”。
- `void xor(BitSet set)`
  这个位集与另一个位集进行逻辑“XOR”。
- `void andNot(BitSet set)`
  清除这个位集中对应另一个位集中设置的所有位。

"Eratosthenes筛子"基准测试

作为位集应用的一个示例，这里给出一个采用"Eratosthenes筛子"算法查找素数的实现（素数是指只能被1和本身整除的数，例如2、3或5，"Eratosthenes筛子"算法是最早发现的用来枚举这些基本数字的方法之一）。这并不是一种查找素数的最好方法，但是由于某种原因，它已经成为测试编译程序性能的一种流行的基准。（这也不是一种最好的基准测试方法，它主要用于测试位操作。）

在此，将尊重这个传统，并给出实现。其程序将计算2～2 000 000之间的所有素数（一共有148 933个素数，或许不打算把它们全部打印出来吧）。

这里并不想深入程序的细节，关键是要遍历一个拥有200万个位的位集。首先将所有的位置为"开"状态，然后，将已知素数的倍数所对应的位都置为"关"状态。经过这个操作保留下来的位对应的就是素数。例13-7是用Java程序设计语言实现的这个算法程序，而例13-8是用C++实现的这个算法程序。

✔ **注释**：尽管筛选并不是一种好的基准测试方法，这里还是对这个算法的两个算法的运行时间进行了测试。下面是在1.66 GHz双核IBM ThinkPad计算机上运行时间的结果，这台计算机内存为2 GB，操作系统为Ubuntu 7.04。

- C++ (g++ 4.1.2): 360毫秒
- Java (Java SE 6): 105毫秒

我们已经对《Java核心技术》7个版本进行了这项测试，在最后的3个版本中，Java轻松地战胜了C++。公平地说，如果有人改变C++优化器的级别，将可以用60毫秒的时间战胜Java。如果程序运行的时间长到触发Hotspot即时编译器时，Java只与C++打个平手。

例13-7 Sieve.java

```
import java.util.*;

/**
 * This program runs the Sieve of Erathostenes benchmark. It computes all primes up to 2,000,000.
 * @version 1.21 2004-08-03
 * @author Cay Horstmann
 */
public class Sieve
{
   public static void main(String[] s)
   {
      int n = 2000000;
      long start = System.currentTimeMillis();
      BitSet b = new BitSet(n + 1);
      int count = 0;
      int i;
      for (i = 2; i <= n; i++)
         b.set(i);
      i = 2;
      while (i * i <= n)
      {
         if (b.get(i))
```

```
            {
               count++;
               int k = 2 * i;
               while (k <= n)
               {
                  b.clear(k);
                  k += i;
               }
            }
            i++;
         }
         while (i <= n)
         {
            if (b.get(i)) count++;
            i++;
         }
         long end = System.currentTimeMillis();
         System.out.println(count + " primes");
         System.out.println((end - start) + " milliseconds");
      }
   }
```

例13-8 Sieve.cpp

```
/**
   @version 1.21 2004-08-03
   @author Cay Horstmann
*/

#include <bitset>
#include <iostream>
#include <ctime>

using namespace std;

int main()
{
   const int N = 2000000;
   clock_t cstart = clock();

   bitset<N + 1> b;
   int count = 0;
   int i;
   for (i = 2; i <= N; i++)
      b.set(i);
   i = 2;
   while (i * i <= N)
   {
      if (b.test(i))
      {
         count++;
         int k = 2 * i;
         while (k <= N)
         {
            b.reset(k);
            k += i;
```

```
            }
        }
        i++;
    }
    while (i <= N)
    {
        if (b.test(i))
            count++;
        i++;
    }

    clock_t cend = clock();
    double millis = 1000.0
        * (cend - cstart) / CLOCKS_PER_SEC;

    cout << count << " primes\n"
        << millis << " milliseconds\n";

    return 0;
}
```

到此为止，Java集合框架的旅程就结束了。正如所看到的，Java类库提供了大量的集合类以适应程序设计的需要。在本书的最后一章，将讨论非常重要的并发程序设计。

第14章 多线程

- ▲ 线程的概念
- ▲ 中断线程
- ▲ 线程状态
- ▲ 线程属性
- ▲ 同步
- ▲ 阻塞队列
- ▲ 线程安全的集合
- ▲ Callable与Future
- ▲ 执行器
- ▲ 同步器
- ▲ 线程与Swing

读者可能已经很熟悉操作系统中的*多任务*（multitasking）：在同一刻运行多个程序的能力。例如，在编辑或下载邮件的同时可以打印文件。今天，人们很可能有单台拥有多个CPU的计算机，但是，并发执行的进程数目并不是由CPU数目制约的。操作系统将CPU的时间片分配给每一个进程，给人并行处理的感觉。

多线程程序在较低的层次上扩展了多任务的概念：一个程序同时执行多个任务。通常，每一个任务称为一个*线程*（thread），它是线程控制的简称。可以同时运行一个以上线程的程序称为*多线程程序*（multithreaded）。

然而，*多进程*与*多线程*有哪些区别呢？本质的区别在于每个进程拥有自己的一整套变量，而线程则共享数据。这听起来似乎有些风险，的确也是这样，在本章稍后将可以看到这个问题。然而，共享变量使线程之间的通信比进程之间的通信更有效、更容易。此外，在有些操作系统中，与进程相比较，线程更"轻量级"，创建、撤销一个线程比启动新进程的开销要小得多。

在实际应用中，多线程非常有用。例如，一个浏览器可以同时下载几幅图片。一个Web服务器需要同时处理几个并发的请求。图形用户界面（GUI）程序用一个独立的线程从宿主操作环境中收集用户界面的事件。本章将介绍如何为Java 应用程序添加多线程能力。

在Java SE 5.0中，多线程发生了重大变化，并增加了大量的类和接口，为大多数应用程序员所需要的多线程机制提供了高质量的实现。本章将介绍Java SE 5.0新增的特性以及类的同步机制，并帮助大家从中做出适当的选择。

温馨提示：多线程可能会变得相当复杂。 本章涵盖了应用程序可能需要的所有工具。尽管如此，对于更复杂的系统级程序设计，建议参看更高级的参考文献，例如：Brian Goetz (Addison-Wesley Professional，2006）的《Java Concurrency in Practice》。

14.1 线程的概念

这里从察看一个没有使用多线程的程序开始。用户很难让它执行多个任务。在对其进行剖析之后，将展示让这个程序运行几个彼此独立的多个线程是很容易的。这个程序采用不断地移

动位置的方式实现球跳动的动画效果，如果发现球碰到墙壁，将进行刷新（见图14-1）。

当点击Start按钮时，程序将从屏幕的左上角弹出一个球，这个球便开始弹跳。Start按钮的处理程序将调用addBall方法。这个方法循环运行1000次move。每调用一次move，球就会移动一点，当碰到墙壁时，球将调整方向，并重新绘制面板。

```
Ball ball = new Ball();
panel.add(ball);
for (int i = 1; i <= STEPS; i++)
{
   ball.move(panel.getBounds());
   panel.paint(panel.getGraphics());
   Thread.sleep(DELAY);
}
```

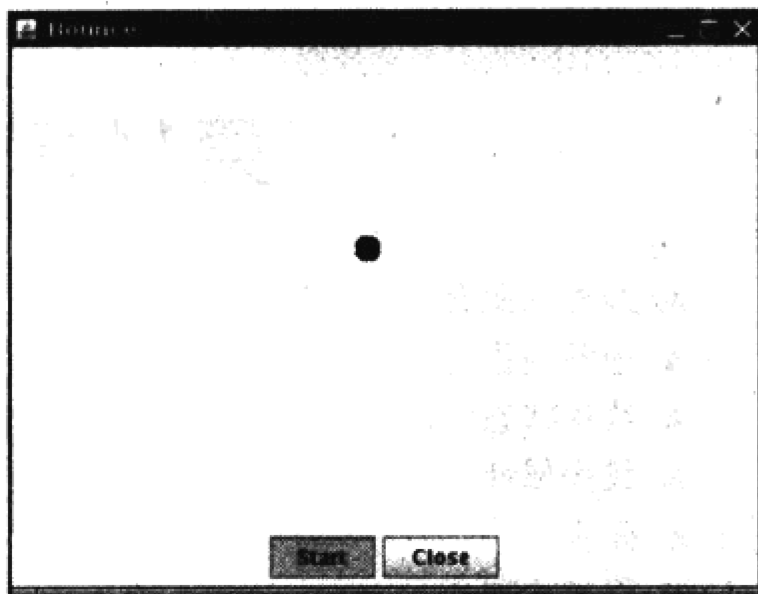


图14-1 使用线程演示跳动的球

Thread类的静态sleep方法将暂停给定的毫秒数。

调用Thread.sleep不会创建一个新线程，sleep是Thread类的静态方法，用于暂停当前线程的活动。

sleep方法可以抛出一个InterruptedException异常。稍后将讨论这个异常以及对它的处理。现在，只是在发生异常时简单地终止弹跳。

如果运行这个程序，球就会自如地来回弹跳，但是，这个程序完全控制了整个应用程序。如果你在球完成1000次弹跳之前已经感到厌倦了，并点击Close 按钮会发现球仍然还在弹跳。在球自己结束弹跳之前无法与程序进行交互。

注释：如果仔细地阅读本节末尾的代码会看到BounceFrame类的addBall方法中有调用

```
comp.paint(comp.getGraphics( ))
```

这一点很奇怪。一般来说，应该调用repaint方法让AWT获得图形上下文并负责绘制。但是，如果试图在这个程序中调用comp.repaint()会发现没有重画面板，这是因为addBall方法完全掌握着控制权。另外，还要注意ball组件扩展于JPanel；这会让擦除背景变得非常容易。接下来的程序将使用一个专门的线程计算球的位置，并会重新使用大家熟悉的repaint和JComponent。

显然，这个程序的性能相当糟糕。人们肯定不愿意让程序用这种方式完成一个非常耗时的工作。毕竟，当通过网络连接读取数据时，阻塞其他任务是经常发生的，有时确实想要中断读取操作。例如，假设下载一幅大图片。当看到一部分图片后，决定不需要或不想再看剩余的部分了，此时，肯定希望能够点击Stop按钮或Back按钮中断下载操作。下一节将介绍如何通过运行一个线程中的关键代码来保持用户对程序的控制权。

例14-1到例14-3给出了这个程序的代码。

例14-1 Bounce.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * Shows an animated bouncing ball.
 * @version 1.33 2007-05-17
 * @author Cay Horstmann
 */
public class Bounce
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new BounceFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * The frame with ball component and buttons.
 */
class BounceFrame extends JFrame
{
   /**
    * Constructs the frame with the component for showing the bouncing ball and Start and
    * Close buttons
    */
   public BounceFrame()
   {
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
      setTitle("Bounce");

      comp = new BallComponent();
      add(comp, BorderLayout.CENTER);
      JPanel buttonPanel = new JPanel();
      addButton(buttonPanel, "Start", new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               addBall();
            }
         });

      addButton(buttonPanel, "Close", new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
```

```
               System.exit(0);
            }
         });
      add(buttonPanel, BorderLayout.SOUTH);
   }

   /**
    * Adds a button to a container.
    * @param c the container
    * @param title the button title
    * @param listener the action listener for the button
    */
   public void addButton(Container c, String title, ActionListener listener)
   {
      JButton button = new JButton(title);
      c.add(button);
      button.addActionListener(listener);
   }

   /**
    * Adds a bouncing ball to the panel and makes it bounce 1,000 times.
    */
   public void addBall()
   {
      try
      {
         Ball ball = new Ball();
         comp.add(ball);

         for (int i = 1; i <= STEPS; i++)
         {
            ball.move(comp.getBounds());
            comp.paint(comp.getGraphics());
            Thread.sleep(DELAY);
         }
      }
      catch (InterruptedException e)
      {
      }
   }

   private BallComponent comp;
   public static final int DEFAULT_WIDTH = 450;
   public static final int DEFAULT_HEIGHT = 350;
   public static final int STEPS = 1000;
   public static final int DELAY = 3;
}
```

例14-2 Ball.java

```
import java.awt.geom.*;

/**
 * A ball that moves and bounces off the edges of a rectangle
 * @version 1.33 2007-05-17
 * @author Cay Horstmann
```

```
   */
public class Ball
{
   /**
    * Moves the ball to the next position, reversing direction if it hits one of the edges
    */
   public void move(Rectangle2D bounds)
   {
      x += dx;
      y += dy;
      if (x < bounds.getMinX())
      {
         x = bounds.getMinX();
         dx = -dx;
      }
      if (x + XSIZE >= bounds.getMaxX())
      {
         x = bounds.getMaxX() - XSIZE;
         dx = -dx;
      }
      if (y < bounds.getMinY())
      {
         y = bounds.getMinY();
         dy = -dy;
      }
      if (y + YSIZE >= bounds.getMaxY())
      {
         y = bounds.getMaxY() - YSIZE;
         dy = -dy;
      }
   }

   /**
    * Gets the shape of the ball at its current position.
    */
   public Ellipse2D getShape()
   {
      return new Ellipse2D.Double(x, y, XSIZE, YSIZE);
   }

   private static final int XSIZE = 15;
   private static final int YSIZE = 15;
   private double x = 0;
   private double y = 0;
   private double dx = 1;
   private double dy = 1;
}
```

例14-3 BallComponent.java

```
import java.awt.*;
import java.util.*;
import javax.swing.*;

/**
 * The component that draws the balls.
```

```
 * @version 1.33 2007-05-17
 * @author Cay Horstmann
 */
public class BallComponent extends JPanel
{
   /**
    * Add a ball to the component.
    * @param b the ball to add
    */
   public void add(Ball b)
   {
      balls.add(b);
   }

   public void paintComponent(Graphics g)
   {
      super.paintComponent(g); // erase background
      Graphics2D g2 = (Graphics2D) g;
      for (Ball b : balls)
      {
         g2.fill(b.getShape());
      }
   }

   private ArrayList<Ball> balls = new ArrayList<Ball>();
}
```

 java.lang.Thread 1.0

- static void sleep(long millis)

  休眠给定的毫秒数。

  参数：millis　　休眠的毫秒数

使用线程给其他任务提供机会

可以将移动球的代码放置在一个独立的线程中，运行这段代码可以提高弹跳球的响应能力。实际上，可以发起多个球，每个球都在自己的线程中运行。另外，AWT的事件分派线程（event dispatch thread）将一直地并行运行，以处理用户界面的事件。由于每个线程都有机会得以运行，所以在球弹跳期间，当用户点击Close按钮时，事件调度线程将有机会关注到这个事件，并处理“关闭”这一动作。

这里用球弹跳代码作为示例，让大家对并发处理有一个视觉印象。通常，人们总会提防长时间的计算。这个计算很可能是某个大框架的一个组成部分，例如，GUI或web框架。无论何时框架调用自身的方法都会很快地返回一个异常。如果需要执行一个比较耗时的任务，应该使用独立的线程。

下面是在一个单独的线程中执行一个任务的简单过程：

1）将任务代码移到实现了Runnable接口的类的run方法中。这个接口非常简单，只有一个方法：

```
public interface Runnable
```

```
{
   void run();
}
```

可以如下所示实现一个类：

```
class MyRunnable implements Runnable
{
   public void run()
   {
      task code
   }
}
```

2）创建一个类对象：

```
Runnable r = new MyRunnable();
```

3）由Runnable创建一个Thread对象：

```
Thread t = new Thread(r);
```

4）启动线程：

```
t.start();
```

要想将弹跳球代码放在一个独立的线程中，只需要实现一个类BallRunnable，然后，将动画代码放在run方法中，如同下面这段代码：

```
class BallRunnable implements Runnable
{
   . . .
   public void run()
   {
      try
      {
         for (int i = 1; i <= STEPS; i++)
         {
            ball.move(component.getBounds());
            component.repaint();
            Thread.sleep(DELAY);
         }
      }
      catch (InterruptedException exception)
      {
      }
   }
   . . .
}
```

此外，需要捕获sleep方法可能抛出的异常InterruptedException。下一节将讨论这个异常。在一般情况下，线程在中断时被终止。因此，当发生InterruptedException异常时，run方法将结束执行。无论何时点击Start按钮，addBall方法都将启动一个新线程（见图14-2）：

```
Ball b = new Ball();
panel.add(b);
Runnable r = new BallRunnable(b, panel);
Thread t = new Thread(r);
t.start();
```

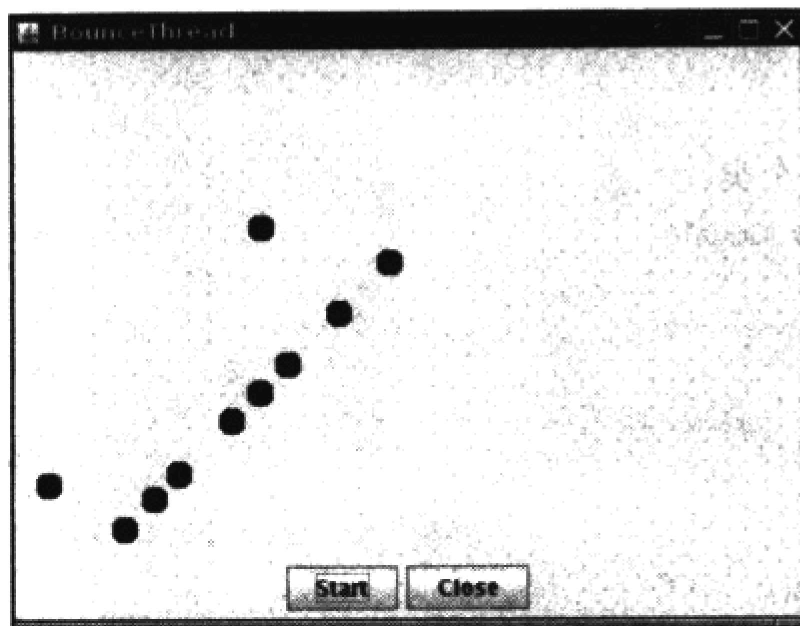


图14-2 运行多线程

这个问题已经讲得差不多了。现在应该知道如何并行运行多个任务了。本章其余部分将阐述如何控制线程之间的交互。

完整的代码在例14-4中。

注释：也可以通过构建一个Thread类的子类定义一个线程，如下所示：

```
class MyThread extends Thread
{
   public void run()
   {
      task code
   }
}
```

然后，构造一个子类的对象，并调用start方法。目前，这种方法已不再推荐。应该从运行机制上减少需要并行运行的任务数量。如果有很多任务，要为每个任务创建一个独立的线程所付出的代价太大了。可以使用线程池来解决这个问题，有关内容请参看第14.9节。

警告：不要调用Thread类或Runnable对象的run方法。直接调用run方法，只会执行同一个线程中的任务，而不会启动新线程。应该调用Thread.start方法。这个方法将创建一个执行run方法的新线程。

例14-4 BounceThread.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * Shows animated bouncing balls.
 * @version 1.33 2007-05-17
 * @author Cay Horstmann
 */
public class BounceThread
```

```
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new BounceFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * A runnable that animates a bouncing ball.
 */
class BallRunnable implements Runnable
{
   /**
    * Constructs the runnable.
    * @aBall the ball to bounce
    * @aPanel the component in which the ball bounces
    */
   public BallRunnable(Ball aBall, Component aComponent)
   {
      ball = aBall;
      component = aComponent;
   }

   public void run()
   {
      try
      {
         for (int i = 1; i <= STEPS; i++)
         {
            ball.move(component.getBounds());
            component.repaint();
            Thread.sleep(DELAY);
         }
      }
      catch (InterruptedException e)
      {
      }
   }

   private Ball ball;
   private Component component;
   public static final int STEPS = 1000;
   public static final int DELAY = 5;
}

/**
 * The frame with panel and buttons.
 */
class BounceFrame extends JFrame
```

```
{
   /**
    * Constructs the frame with the component for showing the bouncing ball and Start and
    * Close buttons
    */
   public BounceFrame()
   {
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
      setTitle("BounceThread");

      comp = new BallComponent();
      add(comp, BorderLayout.CENTER);
      JPanel buttonPanel = new JPanel();
      addButton(buttonPanel, "Start", new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               addBall();
            }
         });

      addButton(buttonPanel, "Close", new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               System.exit(0);
            }
         });
      add(buttonPanel, BorderLayout.SOUTH);
   }

   /**
    * Adds a button to a container.
    * @param c the container
    * @param title the button title
    * @param listener the action listener for the button
    */
   public void addButton(Container c, String title, ActionListener listener)
   {
      JButton button = new JButton(title);
      c.add(button);
      button.addActionListener(listener);
   }

   /**
    * Adds a bouncing ball to the canvas and starts a thread to make it bounce
    */
   public void addBall()
   {
      Ball b = new Ball();
      comp.add(b);
      Runnable r = new BallRunnable(b, comp);
      Thread t = new Thread(r);
      t.start();
   }

   private BallComponent comp;
```

```
    public static final int DEFAULT_WIDTH = 450;
    public static final int DEFAULT_HEIGHT = 350;
    public static final int STEPS = 1000;
    public static final int DELAY = 3;
}
```

java.lang.Thread 1.0

- `Thread(Runnable target)`

  构造一个新线程，用于调用给定target的run（）方法。
- `void start()`

  启动这个线程，将引发调用 run() 方法。这个方法将立即返回，并且新线程将并行运行。
- `void run()`

  调用关联Runnable 的run方法。

java.lang.Runnable 1.0

- `void run()`

  必须覆盖这个方法，并在这个方法中提供所要执行的任务指令。

14.2 中断线程

当线程的run方法执行方法体中最后一条语句后，并经由执行return语句返回时，或者出现了在方法中没有捕获的异常时，线程将终止。在Java的早期版本中，还有一个stop方法，其他线程可以调用它终止线程。但是，这个方法现在已经被弃用了。稍后将讨论“为什么stop方法和suspend方法遭到弃用？”的缘由。

有一种可以强制线程终止的方法。然而，interrupt方法可以用来请求终止线程。

当对一个线程调用interrupt方法时，线程的中断状态将被置位。这是每一个线程都具有的boolean标志。每个线程都应该不时地检查这个标志，以判断线程是否被中断。

要想弄清中断状态是否被置位，首先调用静态的Thread.currentThread方法获得当前线程，然后调用isInterrupted method方法：

```
while (!Thread.currentThread().isInterrupted() && more work to do)
{
   do more work
}
```

但是，如果线程被阻塞，就无法检测中断状态。这是产生InterruptedException异常的地方。当在一个被阻塞的线程（调用sleep或wait）上调用interrupt方法时，阻塞调用将会被Interrupted Exception异常中断。（存在不能被中断的阻塞I/O调用，应该考虑选择可中断的调用。有关细节请参看卷II的第1章和第3章。）

没有任何语言方面的需求要求一个被中断的线程应该终止。中断一个线程不过是引起它的注意。被中断的线程可以决定如何响应中断。某些线程是如此重要以至于应该处理完异常后，继续执行，而不理会中断。但是，更普遍的情况是，线程将简单地将中断作为一个终止的请求。

这种线程的run方法具有如下形式：

```
public void run()
{
   try
   {
      . . .
      while (!Thread.currentThread().isInterrupted() && more work to do)
      {
         do more work
      }
   }
   catch(InterruptedException e)
   {
      // thread was interrupted during sleep or wait
   }
   finally
   {
      cleanup, if required
   }
   // exiting the run method terminates the thread
}
```

如果在每次工作迭代之后都调用sleep方法（或者其他的可中断方法），isInterrupted检测既没有必要也没有用处。如果在中断状态被置位时调用sleep方法，它不会休眠。相反，它将清除这一状态并抛出InterruptedException。因此，如果你的循环调用sleep，不会检测中断状态。相反，要如下所示捕获InterruptedException异常：

```
public void run()
{
   try
   {
      . . .
      while (more work to do)
      {
         do more work
         Thread.sleep(delay);
      }
   }
   catch(InterruptedException e)
   {
      // thread was interrupted during sleep
   }
   finally
   {
      cleanup, if required
   }
   // exiting the run method terminates the thread
}
```

✔ **注释**：有两个非常类似的方法，interrupted和isInterrupted。Interrupted方法是一个静态方法，它检测当前的线程是否被中断。而且，调用interrupted方法会清除该线程的中断状态。另一方面，isInterrupted方法是一个实例方法，可用来检验是否有线程被中断。调用这个方法不会改变中断状态。

在很多发布的代码中会发现InterruptedException异常被抑制在很低的层次上，像这样：

```
void mySubTask()
{
   . . .
   try { sleep(delay); }
   catch (InterruptedException e) {} // DON'T IGNORE!
   . . .
}
```

不要这样做！如果不认为在catch子句中做这一处理有什么好处的话，仍然有两种合理的选择：

- 在catch子句中调用Thread.currentThread().interrupt()来设置中断状态。于是，调用者可以对其进行检测。

  ```
  void mySubTask()
  {
     . . .
     try { sleep(delay); }
     catch (InterruptedException e) { Thread().currentThread().interrupt(); }
     . . .
  }
  ```

- 或者，更好的选择是，用throws InterruptedException标记你的方法，不采用try语句块捕获异常。于是，调用者（或者，最终的run方法）可以捕获这一异常。

  ```
  void mySubTask() throws InterruptedException
  {
     . . .
     sleep(delay);
     . . .
  }
  ```

API java.lang.Thread 1.0

- `void interrupt()`

  向线程发送中断请求。线程的中断状态将被设置为true。如果目前该线程被一个sleep调用阻塞，那么，InterruptedException异常被抛出。
- `static boolean interrupted()`

  测试当前线程（即正在执行这一命令的线程）是否被中断。注意，这是一个静态方法。这一调用会产生副作用——它将当前线程的中断状态重置为false。
- `boolean isInterrupted()`

  测试线程是否被终止。不像静态的中断方法，这一调用不改变线程的中断状态。
- `static Thread currentThread()`

  返回代表当前执行线程的 Thread 对象。

14.3 线程状态

线程可以有如下6种状态：

- New（新生）
- Runnable（可运行）
- Blocked（被阻塞）
- Waiting（等待）
- Timed waiting（计时等待）
- Terminated（被终止）

下一节对每一种状态进行解释。

要确定一个线程的当前状态，可调用getState方法。

14.3.1 新生线程

当用new操作符创建一个新线程时，如new Thread(r)，该线程还没有开始运行。这意味着它的状态是new。当一个线程处于新生状态时，程序还没有开始运行线程中的代码。在线程运行之前还有一些簿记工作要做。

14.3.2 可运行线程

一旦调用start方法，线程处于runnable状态。一个可运行的线程可能正在运行也可能没有运行，这取决于操作系统给线程提供运行的时间。（Java的规范说明没有将它作为一个单独状态。一个正在运行中的线程仍然处于可运行状态。）

一旦一个线程开始运行，它不必始终保持运行。事实上，运行中的线程被中断，目的是为了让其他线程获得运行机会。线程调度的细节依赖于操作系统提供的服务。抢占式调度系统给每一个可运行线程一个时间片来执行任务。当时间片用完，操作系统剥夺该线程的运行权，并给另一个线程运行机会（见图14-4）。当选择下一个线程时，操作系统考虑线程的优先级——更多的内容见第14.4.1节。

现在所有的桌面以及服务器操作系统都使用抢占式调度。但是，像手机这样的小型设备可能使用协作式调度。在这样的设备中，一个线程只有在调用yield方法、或者被阻塞或等待时，线程才失去控制权。

在具有多个处理器的机器上，每一个处理器运行一个线程，可以有多个线程并行运行。当然，如果线程的数目多于处理器的数目，调度器依然采用时间片机制。

记住，在任何给定时刻，一个可运行的线程可能正在运行也可能没有运行（这就是为什么将这个状态称为可运行而不是运行）。

14.3.3 被阻塞线程和等待线程

当线程处于被阻塞或等待状态时，它暂时不活动。它不运行任何代码且消耗最少的资源。直到线程调度器重新激活它。细节取决于它是怎样达到非活动状态的。

- 当一个线程试图获取一个内部的对象锁（而不是java.util.concurrent库中的锁），而该锁被其他线程持有，则该线程进入阻塞状态（我们在第14.5.3节讨论java.util.concurrent锁，在第14.5.5节讨论内部对象锁）。当所有其他线程释放该锁，并且线程调度器允许本线程持有它的时候，该线程将变成非阻塞状态。

- 当线程等待另一个线程通知调度器一个条件时，它自己进入等待状态。我们在第14.5.4节来讨论条件。在调用Object.wait方法或Thread.join方法，或者是等待java.util.concurrent库中的Lock或Condition时，就会出现这种情况。实际上，被阻塞状态与等待状态是有很大不同的。
- 有几个方法有一个超时参数。调用它们导致线程进入计时等待（timed waiting）状态。这一状态将一直保持到超时期满或者接收到适当的通知。带有超时参数的方法有Thread.sleep和Object.wait、Thread.join、Lock.tryLock以及Condition.await的计时版。

图14-3展示了线程可以具有的状态以及从一个状态到另一个状态可能的转换。当一个线程被阻塞或等待时（或终止时），另一个线程被调度为运行状态。当一个线程被重新激活（例如，因为超时期满或成功地获得了一个锁），调度器检查它是否具有比当前运行线程更高的优先级。如果是这样，调度器从当前运行线程中挑选一个，剥夺其运行权，选择一个新的线程运行。

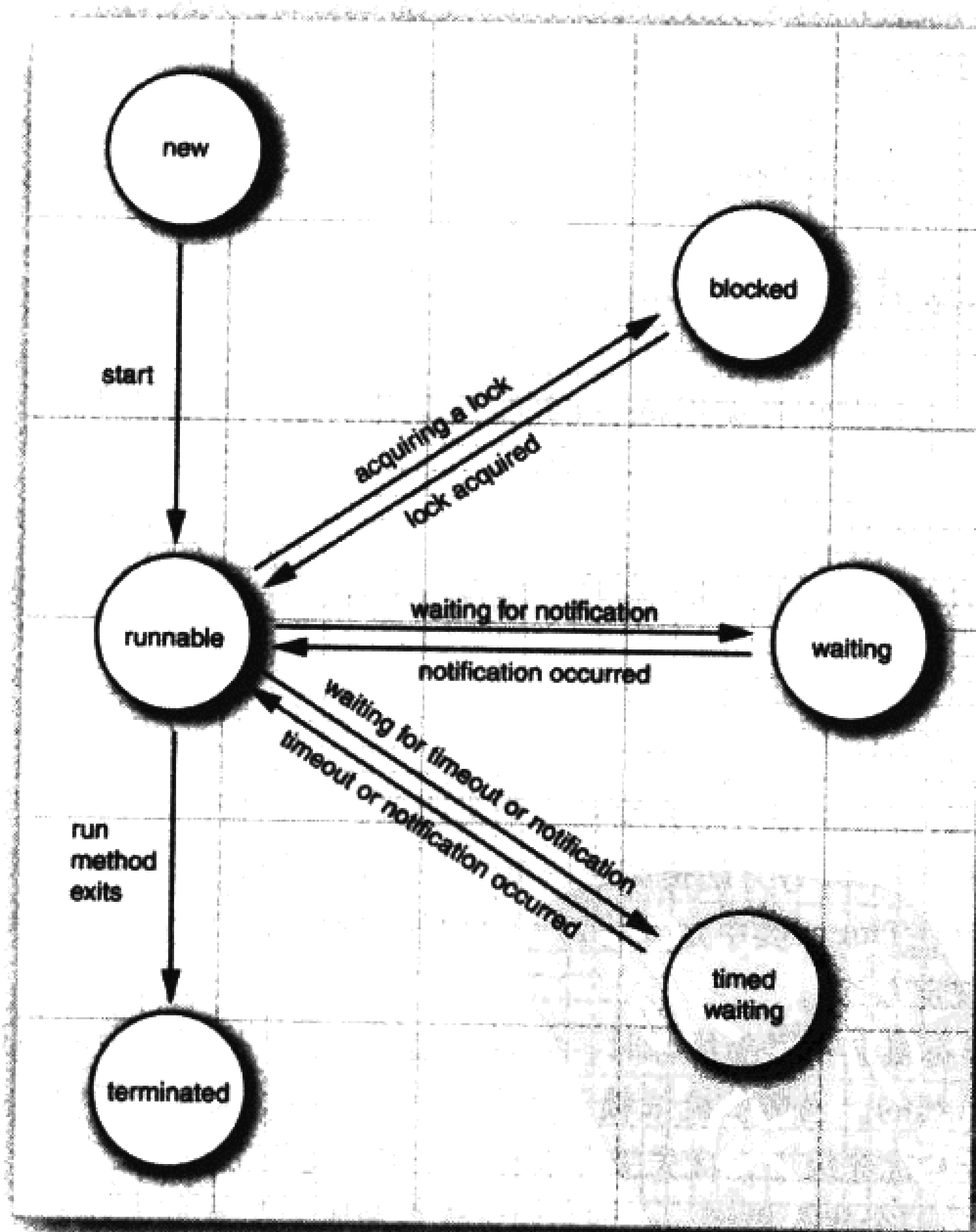


图14-3 线程状态

14.3.4 被终止的线程

线程因如下两个原因之一而被终止：

- 因为run方法正常退出而自然死亡。
- 因为一个没有捕获的异常终止了run方法而意外死亡。

特别是，可以调用线程的stop方法杀死一个线程。该方法抛出ThreadDeath 错误对象，由此杀死线程。但是，stop方法已过时，不要在自己的代码中调用它。

API java.lang.Thread 1.0

- `void join()`
  等待终止指定的线程。
- `void join(long millis)`
  等待指定的线程死亡或者经过指定的毫秒数。
- `Thread.State getState() 5.0`
  得到这一线程的状态；NEW、RUNNABLE、BLOCKED、WAITING、TIMED_WAITING 或TERMINATED之一。
- `void stop()`
  停止该线程。这一方法已过时。
- `void suspend()`
  暂停这一线程的执行。这一方法已过时。
- `void resume()`
  恢复线程。这一方法仅仅在调用suspend()之后调用。这一方法已过时。

14.4 线程属性

下面将讨论线程的各种属性，其中包括：线程优先级、守护线程、线程组以及处理未捕获异常的处理器。

14.4.1 线程优先级

在Java程序设计语言中，每一个线程有一个*优先级*。默认情况下，一个线程继承它的父线程的优先级。可以用setPriority方法提高或降低任何一个线程的优先级。可以将优先级设置为在MIN_PRIORITY（在Thread类中定义为1）与MAX_PRIORITY（定义为10）之间的任何值。NORM_PRIORITY被定义为5。

每当线程调度器有机会选择新线程时，它首先选择具有较高优先级的线程。但是，线程优先级是*高度依赖于系统的*。当虚拟机依赖于宿主机平台的线程实现机制时，Java线程的优先级被映射到宿主机平台的优先级上，优先级个数也许更多，也许更少。

例如，Windows有7个优先级别。一些Java优先级将映射到相同的操作系统优先级。在Sun为Linux提供的Java虚拟机，线程的优先级被忽略——所有线程具有相同的优先级。

初级程序员常常过度使用线程优先级。为优先级而烦恼是事出有因的。不要将程序构建为功能的正确性依赖于优先级。

警告：如果确实要使用优先级，应该避免初学者常犯的一个错误。如果有几个高优先级的线程没有进入非活动状态，低优先级的线程可能永远也不能执行。每当调度器决定运行一个新线程时，首先会在具有高优先级的线程中进行选择，尽管这样会使低优先级的线程完全饿死。

API java.lang.Thread 1.0

- void setPriority(int newPriority)

  设置线程的优先级。优先级必须在 Thread.MIN_PRIORITY 与Thread.MAX_PRIORITY之间。一般使用 Thread.NORM_PRIORITY 优先级。
- static int MIN_PRIORITY

  线程的最小优先级。最小优先级的值为1。
- static int NORM_PRIORITY

  线程的默认优先级。缺省优先级为5。
- static int MAX_PRIORITY

  线程的最高优先级。最高优先级的值为10 。
- static void yield()

  导致当前执行线程处于让步状态。如果有其他的可运行线程具有至少与此线程同样高的优先级，那么这些线程接下来会被调度。注意，这是一个静态方法。

14.4.2 守护线程

可以通过调用

```
t.SetDaemon(true);
```

将线程转换为守护线程（daemon thread）。这样一个线程没有什么神奇。守护线程的惟一用途是为其他线程提供服务。计时线程就是一个例子，它定时地发送"时间嘀嗒"信号给其他线程或清空过时的高速缓存项的线程。当只剩下守护线程时，虚拟机就退出了，由于如果只剩下守护线程，就没必要继续运行程序了。

守护线程有时会被初学者错误地使用，他们不打算考虑关机（shutdown）动作。但是，这是很危险的。守护线程应该永远不去访问固有资源，如文件、数据库，因为它会在任何时候甚至在一个操作的中间发生中断。

API java.lang.Thread 1.0

- void setDaemon(boolean isDaemon)

  标识该线程为守护线程或用户线程。这一方法必须在线程启动之前调用。

14.4.3 未捕获异常处理器

线程的run方法不能抛出任何被检测的异常，但是，不被检测的异常会导致线程终止。在这种情况下，线程就死亡了。

但是，不需要任何catch子句来处理可以被传播的异常。相反，就在线程死亡之前，异常被传递到一个用于未捕获异常的处理器。

该处理器必须属于一个实现Thread.UncaughtExceptionHandler接口的类。这个接口只有一个方法。

```
void uncaughtException(Thread t, Throwable e)
```

从Java SE 5.0起，可以用setUncaughtExceptionHandler方法为任何线程安装一个处理器。也可以用Thread类的静态方法setDefaultUncaughtExceptionHandler为所有线程安装一个默认的处理器。替换处理器可以使用日志API发送未捕获异常的报告到日志文件。

如果不安装默认的处理器，默认的处理器为空。但是，如果不为独立的线程安装处理器，此时的处理器就是该线程的ThreadGroup对象。

> **注释**：线程组是一个可以统一管理的线程集合。默认情况下，创建的所有线程属于相同的线程组，但是，也可能会建立其他的组。从Java SE 5.0起引入了更好的特性用于线程集合的操作。不要在自己的程序中使用线程组。

ThreadGroup类实现Thread.UncaughtExceptionHandler接口。它的uncaughtException方法做如下操作：

1）如果该线程组有父线程组，那么父线程组的uncaughtException方法被调用。

2）否则，如果Thread.getDefaultExceptionHandler方法返回一个非空的处理器，则调用该处理器。

3）否则，如果Throwable是ThreadDeath的一个实例，什么都不做。

4）否则，线程的名字以及Throwable的栈踪迹被输出到System.err上。

这是你在程序中肯定看到过许多次的栈踪迹。

API java.lang.Thread 1.0

- `static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler handler) 5.0`
- `static Thread.UncaughtExceptionHandler getDefaultUncaughtExceptionHandler() 5.0`

  设置或获取未捕获异常的默认处理器。
- `void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler handler) 5.0`
- `Thread.UncaughtExceptionHandler getUncaughtExceptionHandler() 5.0`

  设置或获取未捕获异常的处理器。如果没有安装处理器，则将线程组对象作为处理器。

API java.lang.Thread.UncaughtExceptionHandler 5.0

- `void uncaughtException(Thread t, Throwable e)`

  当一个线程因未捕获异常而终止，按规定要将客户报告记录到日志中。

  参数：t　　由于未捕获异常而终止的线程

  　　　e　　未捕获的异常对象

java.lang.ThreadGroup 1.0

- void uncaughtException(Thread t, Throwable e)

  如果有父线程组，调用父线程组的这一方法；或者，如果Thread类有默认处理器，调用该处理器，否则，输出栈踪迹到标准错误流上（但是，如果e是一个ThreadDeath对象，栈踪迹是被禁用的。ThreadDeath对象由stop方法产生，而该方法已经过时）。

14.5 同步

在大多数实际的多线程应用中，两个或两个以上的线程需要共享对同一数据的存取。如果两个线程存取相同的对象，并且每一个线程都调用了一个修改该对象状态的方法，将会发生什么呢？可以想像，线程彼此踩了对方的脚。根据各线程访问数据的次序，可能会产生讹误的对象。这样一个情况通常称为*竞争条件*（race condition）。

14.5.1 竞争条件的一个例子

为了避免多线程引起的对共享数据的讹误，必须学习如何*同步存取*。在本节中，你会看到如果没有使用同步会发生什么。在下一节中，将会看到如何同步数据存取。

在下面的测试程序中，模拟一个有若干账户的银行。随机地生成在这些账户之间转移钱款的交易。每一个账户有一个线程。每一笔交易中，会从线程所服务的账户中随机转移一定数目的钱款到另一个随机账户。

模拟代码非常直观。我们有具有transfer方法的Bank类。该方法从一个账户转移一定数目的钱款到另一个账户（还没有考虑负的账户余额）。如下是Bank类的transfer方法的代码。

```
public void transfer(int from, int to, double amount)
   // CAUTION: unsafe when called from multiple threads
{
   System.out.print(Thread.currentThread());
   accounts[from] -= amount;
   System.out.printf(" %10.2f from %d to %d", amount, from, to);
   accounts[to] += amount;
   System.out.printf(" Total Balance: %10.2f%n", getTotalBalance());
}
```

这里是TransferRunnable类的代码。它的run方法不断地从一个固定的银行账户取出钱款。在每一次迭代中，run方法随机选择一个目标账户和一个随机账户，调用bank对象的transfer方法，然后睡眠。

```
class TransferRunnable implements Runnable
{
   . . .
   public void run()
   {
      try
      {
         int toAccount = (int) (bank.size() * Math.random());
         double amount = maxAmount * Math.random();
         bank.transfer(fromAccount, toAccount, amount);
```

```
            Thread.sleep((int) (DELAY * Math.random()));
         }
         catch(InterruptedException e) {}
      }
   }
```

当这个模拟程序运行时，不清楚在某一时刻某一银行账户中有多少钱。但是，知道所有账户的总金额应该保持不变，因为所做的一切不过是从一个账户转移钱款到另一个账户。

在每一次交易的结尾，transfer方法重新计算总值并打印出来。

本程序永远不会结束。只能按CTRL+C来终止这个程序。

下面是典型的输出：

```
. . .
Thread[Thread-11,5,main]    588.48 from 11 to 44 Total Balance:  100000.00
Thread[Thread-12,5,main]    976.11 from 12 to 22 Total Balance:  100000.00
Thread[Thread-14,5,main]    521.51 from 14 to 22 Total Balance:  100000.00
Thread[Thread-13,5,main]    359.89 from 13 to 81 Total Balance:  100000.00
. . .
Thread[Thread-36,5,main]    401.71 from 36 to 73 Total Balance:   99291.06
Thread[Thread-35,5,main]    691.46 from 35 to 77 Total Balance:   99291.06
Thread[Thread-37,5,main]     78.64 from 37 to 3 Total Balance:   99291.06
Thread[Thread-34,5,main]    197.11 from 34 to 69 Total Balance:   99291.06
Thread[Thread-36,5,main]     85.96 from 36 to 4 Total Balance:   99291.06
. . .
Thread[Thread-4,5,main]Thread[Thread-33,5,main]      7.31 from 31 to 32 Total Balance:
99979.24
    627.50 from 4 to 5 Total Balance:   99979.24
. . .
```

正如前面所示，出现了错误。在最初的交易中，银行的余额保持在$100 000，这是正确的，因为共100个账户，每个账户$1 000。但是，过一段时间，余额总量有轻微的变化。 当运行这个程序的时候，会发现有时很快就出错了，有时很长的时间后余额发生混乱。这样的状态不会带来信任感，人们很可能不愿意将辛苦挣来得钱存到这个银行。

例14-5到例14-7中的程序提供了完全的源代码。看看是否可以从代码中找出问题。下一节将解说其中神秘。

例14-5 UnsynchBankTest.java

```
/**
 * This program shows data corruption when multiple threads access a data structure.
 * @version 1.30 2004-08-01
 * @author Cay Horstmann
 */
public class UnsynchBankTest
{
   public static void main(String[] args)
   {
      Bank b = new Bank(NACCOUNTS, INITIAL_BALANCE);
      int i;
      for (i = 0; i < NACCOUNTS; i++)
      {
         TransferRunnable r = new TransferRunnable(b, i, INITIAL_BALANCE);
```

• void uncaughtException(Thread t, Throwable e)

```
            Thread t = new Thread(r);
            t.start();
         }
      }

   public static final int NACCOUNTS = 100;
   public static final double INITIAL_BALANCE = 1000;
}
```

例14-6 Bank.java

```
/**
 * A bank with a number of bank accounts.
 * @version 1.30 2004-08-01
 * @author Cay Horstmann
 */
public class Bank
{
   /**
    * Constructs the bank.
    * @param n the number of accounts
    * @param initialBalance the initial balance for each account
    */
   public Bank(int n, double initialBalance)
   {
      accounts = new double[n];
      for (int i = 0; i < accounts.length; i++)
         accounts[i] = initialBalance;
   }

   /**
    * Transfers money from one account to another.
    * @param from the account to transfer from
    * @param to the account to transfer to
    * @param amount the amount to transfer
    */
   public void transfer(int from, int to, double amount)
   {
      if (accounts[from] < amount) return;
      System.out.print(Thread.currentThread());
      accounts[from] -= amount;
      System.out.printf(" %10.2f from %d to %d", amount, from, to);
      accounts[to] += amount;
      System.out.printf(" Total Balance: %10.2f%n", getTotalBalance());
   }

   /**
    * Gets the sum of all account balances.
    * @return the total balance
    */
   public double getTotalBalance()
   {
      double sum = 0;

      for (double a : accounts)
         sum += a;
```

```
        return sum;
    }

    /**
     * Gets the number of accounts in the bank.
     * @return the number of accounts
     */
    public int size()
    {
        return accounts.length;
    }

    private final double[] accounts;
}
```

例14-7 TransferRunnable.java

```
/**
 * A runnable that transfers money from an account to other accounts in a bank.
 * @version 1.30 2004-08-01
 * @author Cay Horstmann
 */
public class TransferRunnable implements Runnable
{
    /**
     * Constructs a transfer runnable.
     * @param b the bank between whose account money is transferred
     * @param from the account to transfer money from
     * @param max the maximum amount of money in each transfer
     */
    public TransferRunnable(Bank b, int from, double max)
    {
        bank = b;
        fromAccount = from;
        maxAmount = max;
    }

    public void run()
    {
        try
        {
            while (true)
            {
                int toAccount = (int) (bank.size() * Math.random());
                double amount = maxAmount * Math.random();
                bank.transfer(fromAccount, toAccount, amount);
                Thread.sleep((int) (DELAY * Math.random()));
            }
        }
        catch (InterruptedException e)
        {
        }
    }

    private Bank bank;
```

```
private int fromAccount;
private double maxAmount;
private int DELAY = 10;
}
```

14.5.2 详解竞争条件

上一节中运行了一个程序，其中有几个线程更新银行账户余额。一段时间之后，错误不知不觉地出现了，总额要么增加，要么变少。当两个线程试图同时更新同一个账户的时候，这个问题就出现了。假定两个线程同时执行指令

```
accounts[to] += amount;
```

问题在于这不是原子操作。该指令可能被处理如下：

1）将accounts[to]加载到寄存器。

2）增加amount。

3）将结果写回accounts[to]。

现在，假定第1个线程执行步骤1和2，然后，它被剥夺了运行权。假定第2个线程被唤醒并修改了accounts数组中的同一项。然后，第1个线程被唤醒并完成其第3步。

这样，这一动作擦去了第二个线程所做的更新。于是，总金额不再正确。（见图14-4。）

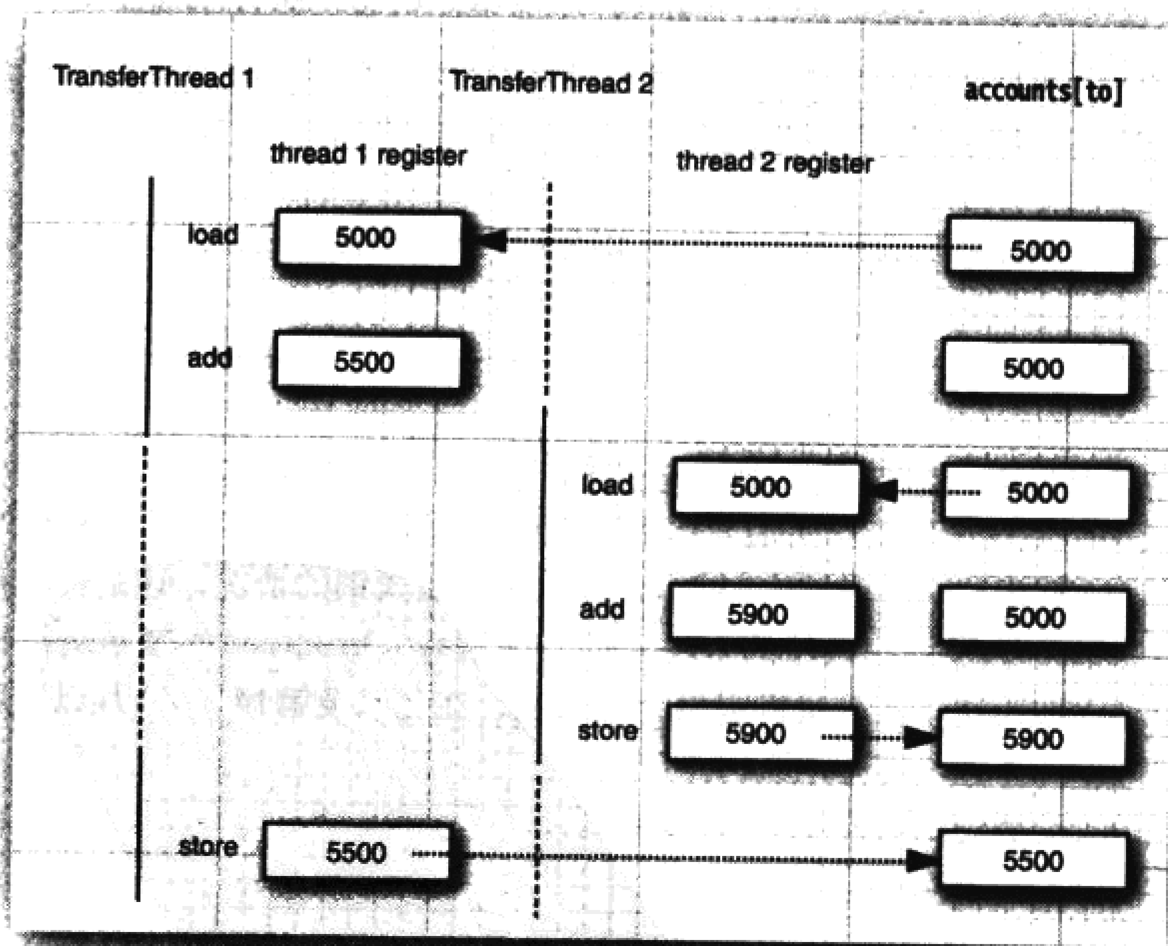


图14-4 同时被两个线程访问

我们的测试程序检测到这一讹误。（当然，如果线程在运行这一测试时被中断，也有可能会出现失败警告！）



注释：可以具体看一下执行我们的类中的每一个语句的虚拟机的字节码。运行命令

```
javap -c -v Bank
```

对Bank.class文件进行反编译。例如，代码行

```
accounts[to] += amount;
```

被转换为下面的字节码：

```
aload_0
getfield        #2; //Field accounts:[D
iload_2
dup2
daload
dload_3
dadd
dastore
```

这些代码的含义无关紧要。重要的是增值命令是由几条指令组成的，执行它们的线程可以在任何一条指令点上被中断。

出现这一讹误的可能性有多大呢？这里通过将打印语句和更新余额的语句交织在一起执行，增加了发生这种情况的机会。

如果删除打印语句，讹误的风险会降低一点，因为每个线程在再次睡眠之前所做的工作很少，调度器在计算过程中剥夺线程的运行权可能性很小。但是，讹误的风险并没有完全消失。如果在负载很重的机器上运行许多线程，那么，即使删除了打印语句，程序依然会出错。这种错误可能会几分钟、几小时或几天出现一次。坦白地说，对程序员而言，很少有比无规律出现错误更糟的事情了。

真正的问题是transfer方法的执行过程中可能会被中断。如果能够确保线程在失去控制之前方法运行完成，那么银行账户对象的状态永远不会出现讹误。

14.5.3 锁对象

从 Java SE 5.0开始，有两种机制防止代码块受并发访问的干扰。Java语言提供一个synchronized关键字达到这一目的，并且Java SE 5.0引入了ReentrantLock类。synchronized关键字自动提供一个锁以及相关的"条件"，对于大多数需要显式锁的情况，这是很便利的。但是，我们相信在读者分别阅读了锁和条件的内容之后，理解synchronized关键字是很轻松的事情。java.util.concurrent框架为这些基础机制提供独立的类，在此以及第14.5.4节加以解释这个内容。读者理解了这些构建块之后，将讨论第14.5.5节。

用ReentrantLock保护代码块的基本结构如下：

```
myLock.lock(); // a ReentrantLock object
try
{
   critical section
}
finally
{
   myLock.unlock(); // make sure the lock is unlocked even if an exception is thrown
}
```

这一结构确保任何时刻只有一个线程进入临界区。一旦一个线程封锁了锁对象，其他任何线程都无法通过lock语句。当其他线程调用lock时，它们被阻塞，直到第一个线程释放锁对象。

警告：把解锁操作括在finally子句之内是至关重要的。如果在临界区的代码抛出异常，锁必须被释放。否则，其他线程将永远阻塞。

让我们使用一个锁来保护Bank类的 transfer 方法。

```
public class Bank
{
   public void transfer(int from, int to, int amount)
   {
      bankLock.lock();
      try
      {
         System.out.print(Thread.currentThread());
         accounts[from] -= amount;
         System.out.printf(" %10.2f from %d to %d", amount, from, to);
         accounts[to] += amount;
         System.out.printf(" Total Balance: %10.2f%n", getTotalBalance());
      }
      finally
      {
         bankLock.unlock();
      }
   }
   . . .
   private Lock bankLock = new ReentrantLock(); // ReentrantLock implements the Lock interface
}
```

假定一个线程调用transfer，在执行结束前被剥夺了运行权。假定第二个线程也调用transfer，由于第二个线程不能获得锁，将在调用lock方法时被阻塞。它必须等待第一个线程完成transfer方法的执行之后才能再度被激活。当第一个线程释放锁时，那么第二个线程才能开始运行（见图14-5）。

尝试一下。添加加锁代码到transfer方法并且再次运行程序。你可以永远运行它，而银行的余额不会出现讹误。

注意每一个Bank对象有自己的ReentrantLock对象。如果两个线程试图访问同一个Bank对象，那么锁以串行方式提供服务。但是，如果两个线程访问不同的Bank对象，每一个线程得到不同的锁对象，两个线程都不会发生阻塞。本该如此，因为线程在操纵不同的Bank实例的时候，线程之间不会相互影响。

锁是可重入的，因为线程可以重复地获得已经持有的锁。锁保持一个持有计数（hold count）来跟踪对lock方法的嵌套调用。线程在每一次调用lock都要调用unlock来释放锁。由于这一特性，被一个锁保护的代码可以调用另一个使用相同的锁的方法。

例如，transfer方法调用getTotalBalance方法，这也会封锁bankLock对象，此时bankLock对象的持有计数为2。当getTotalBalance方法退出的时候，持有计数变回1。当transfer方法退出的时候，持有计数变为0。线程释放锁。

通常，可能想要保护需若干个操作来更新或检查共享对象的代码块。要确保这些操作完成

后，另一个线程才能使用相同对象。

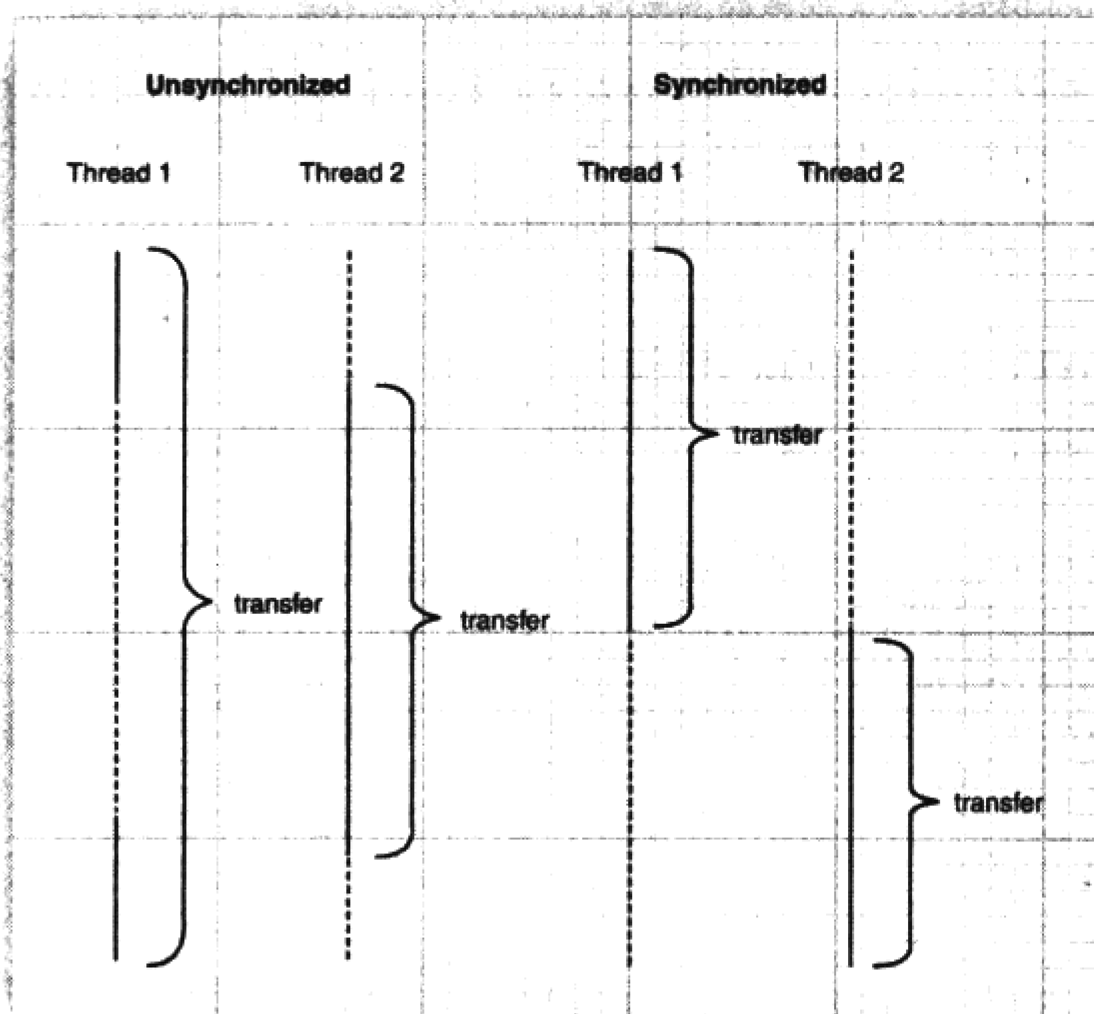


图14-5 非同步线程与同步线程的比较

警告：要留心临界区中的代码，不要因为异常的抛出而跳出了临界区。如果在临界区代码结束之前抛出了异常，finally子句将释放锁，但会使对象可能处于一种受损状态。

API `java.util.concurrent.locks.Lock 5.0`

- `void lock()`

  获取这个锁；如果锁同时被另一个线程拥有则发生阻塞。

- `void unlock()`

  释放这个锁。

API `java.util.concurrent.locks.ReentrantLock 5.0`

- `ReentrantLock()`

  构建一个可以被用来保护临界区的可重入锁。

- `ReentrantLock(boolean fair)`

  构建一个带有公平策略的锁。一个公平锁偏爱等待时间最长的线程。但是，这一公平的保证将大大降低性能。所以，默认情况下，锁没有被强制为公平的。

警告：听起来公平锁更合理一些，但是使用公平锁比使用常规锁要慢很多。只有当你确实了解自己要做什么并且对于你要解决的问题有一个特定的理由必须使用公平锁的时候，才可以使用公平锁。即使使用公平锁，也无法确保线程调度器是公平的。如果线程调度器选择忽略一个线程，而该线程为了这个锁已经等待了很长时间，那么就没有机会公平地处理这个锁了。

14.5.4 条件对象

通常，线程进入临界区，却发现在某一条件满足之后它才能执行。要使用一个条件对象来管理那些已经获得了一个锁但是却不能做有用工作的线程。在这一节里，我们介绍Java库中条件对象的实现。(由于历史的原因，条件对象经常被称为*条件变量*（conditional variable)。)

现在来细化银行的模拟程序。我们避免选择没有足够资金的账户作为转出账户。注意不能使用下面这样的代码：

```
if (bank.getBalance(from) >= amount)
   bank.transfer(from, to, amount);
```

当前线程完全有可能在成功地完成测试，且在调用transfer方法之前将被中断。

```
if (bank.getBalance(from) >= amount)
      // thread might be deactivated at this point
   bank.transfer(from, to, amount);
```

在线程再次运行前，账户余额可能已经低于提款金额。必须确保没有其他线程在本检查余额与转账活动之间修改余额。通过使用锁来保护检查与转账动作来做到这一点：

```
public void transfer(int from, int to, int amount)
{
   bankLock.lock();
   try
   {
      while (accounts[from] < amount)
      {
         // wait
         . . .
      }
      // transfer funds
      . . .
   }
   finally
   {
      bankLock.unlock();
   }
}
```

现在，当账户中没有足够的余额时，应该做什么呢？等待直到另一个线程向账户中注入了资金。但是，这一线程刚刚获得了对bankLock的排它性访问，因此别的线程没有进行存款操作的机会。这就是为什么我们需要条件对象的原因。

一个锁对象可以有一个或多个相关的条件对象。你可以用newCondition方法获得一个条件对象。习惯上给每一个条件对象命名为可以反映它所表达的条件的名字。例如，在此设置一个

条件对象来表达“余额充足”条件。

```
class Bank
{
   public Bank()
   {
      . . .
      sufficientFunds = bankLock.newCondition();
   }
   . . .
   private Condition sufficientFunds;
}
```

如果transfer方法发现余额不足，它调用

```
sufficientFunds.await();
```

当前线程现在被阻塞了，并放弃了锁。我们希望这样可以使得另一个线程可以进行增加账户余额的操作。

等待获得锁的线程和调用await方法的线程存在本质上的不同。一旦一个线程调用await方法，它进入该条件的等待集。当锁可用时，该线程不能马上解除阻塞。相反，它处于阻塞状态，直到另一个线程调用同一条件上的signalAll方法时为止。当另一个线程转账时，它应该调用

```
sufficientFunds.signalAll();
```

这一调用重新激活因为这一条件而等待的所有线程。当这些线程从等待集当中移出时，它们再次成为可运行的，调度器将再次激活它们。同时，它们将试图重新进入该对象。一旦锁成为可用的，它们中的某个将从await调用返回，获得该锁并从被阻塞的地方继续执行。

此时，线程应该再次测试该条件。由于无法确保该条件被满足——signalAll方法仅仅是通知正在等待的线程：此时有可能已经满足条件，值得再次去检测该条件。



注释：通常，对await的调用应该在如下形式的循环体中

```
while (!(ok to proceed))
   condition.await();
```

至关重要的是最终需要某个其他线程调用signalAll方法。当一个线程调用await时，它没有办法重新激活自身。它寄希望于其他线程。如果没有其他线程来重新激活等待的线程，它就永远不再运行了。这将导致令人不快的*死锁*（deadlock）现象。如果所有其他线程被阻塞，最后一个活动线程在解除其他线程的阻塞状态之前就调用await方法，那么它也被阻塞。没有任何线程可以解除其他线程的阻塞，那么该程序就挂起了。

应该何时调用signalAll呢？经验上讲，在对象的状态有利于等待线程的方向改变时调用signalAll。例如，当一个账户余额发生改变时，等待的线程会应该有机会检查余额。在例子中，当完成了转账时，调用signalAll方法。

```
public void transfer(int from, int to, int amount)
{
   bankLock.lock();
   try
   {
      while (accounts[from] < amount)
```

```
         sufficientFunds.await();
      // transfer funds
      . . .
      sufficientFunds.signalAll();
   }
   finally
   {
      bankLock.unlock();
   }
}
```

注意调用signalAll不会立即激活一个等待线程。它仅仅解除等待线程的阻塞，以便这些线程可以在当前线程退出同步方法之后，通过竞争实现对对象的访问。

另一个方法signal，则是随机解除等待集中某个线程的阻塞状态。这比解除所有线程的阻塞更加有效，但也存在危险。如果随机选择的线程发现自己仍然不能运行，那么它再次被阻塞。如果没有其他线程再次调用signal，那么系统就死锁了。

警告：当一个线程拥有某个条件的锁时，它仅仅可以在该条件上调用await、signalAll或signal方法。

如果你运行例14-8中的程序，会注意到没有出现任何错误。总余额永远是$100 000。没有任何账户曾出现负的余额（但是，你还是需要按下CTRL+C键来终止程序）。你可能还注意到这个程序运行起来稍微有些慢——这是为同步机制中的簿记操作所付出的代价。

实际上，正确地使用条件是富有挑战性的。在开始实现自己的条件对象之前，应该考虑使用“同步器”中描述的结构。

例14-8 Bank.java

```
import java.util.concurrent.locks.*;

/**
 * A bank with a number of bank accounts that uses locks for serializing access.
 * @version 1.30 2004-08-01
 * @author Cay Horstmann
 */
public class Bank
{
   /**
    * Constructs the bank.
    * @param n the number of accounts
    * @param initialBalance the initial balance for each account
    */
   public Bank(int n, double initialBalance)
   {
      accounts = new double[n];
      for (int i = 0; i < accounts.length; i++)
         accounts[i] = initialBalance;
      bankLock = new ReentrantLock();
      sufficientFunds = bankLock.newCondition();
   }

   /**
```

```
    * Transfers money from one account to another.
    * @param from the account to transfer from
    * @param to the account to transfer to
    * @param amount the amount to transfer
    */
   public void transfer(int from, int to, double amount) throws InterruptedException
   {
      bankLock.lock();
      try
      {
         while (accounts[from] < amount)
            sufficientFunds.await();
         System.out.print(Thread.currentThread());
         accounts[from] -= amount;
         System.out.printf(" %10.2f from %d to %d", amount, from, to);
         accounts[to] += amount;
         System.out.printf(" Total Balance: %10.2f%n", getTotalBalance());
         sufficientFunds.signalAll();
      }
      finally
      {
         bankLock.unlock();
      }
   }

   /**
    * Gets the sum of all account balances.
    * @return the total balance
    */
   public double getTotalBalance()
   {
      bankLock.lock();
      try
      {
         double sum = 0;

         for (double a : accounts)
            sum += a;

         return sum;
      }
      finally
      {
         bankLock.unlock();
      }
   }

   /**
    * Gets the number of accounts in the bank.
    * @return the number of accounts
    */
   public int size()
   {
      return accounts.length;
   }

```

```
    private final double[] accounts;
    private Lock bankLock;
    private Condition sufficientFunds;
}
```

API java.util.concurrent.locks.Lock 5.0

- `Condition newCondition()`

  返回一个与该锁相关的条件对象。

API java.util.concurrent.locks.Condition 5.0

- `void await()`

  将该线程放到条件的等待集中。
- `void signalAll()`

  解除该条件的等待集中的所有线程的阻塞状态。
- `void signal()`

  从该条件的等待集中随机地选择一个线程，解除其阻塞状态。

14.5.5 synchronized关键字

在前面一节中，介绍了如何使用Lock和Condition对象。在进一步深入之前，总结一下有关锁和条件的关键之处:

- 锁用来保护代码片段，任何时刻只能有一个线程执行被保护的代码。
- 锁可以管理试图进入被保护代码段的线程。
- 锁可以拥有一个或多个相关的条件对象。
- 每个条件对象管理那些已经进入被保护的代码段但还不能运行的线程。

Lock和Condition接口被添加到Java SE 5.0中，这也向程序设计人员提供了高度的封锁控制。然而，大多数情况下，并不需要那样的控制，并且可以使用一种嵌入到Java语言内部的机制。从1.0版开始，Java中的每一个对象都有一个内部锁。如果一个方法用synchronized 关键字声明，那么对象的锁将保护整个方法。也就是说，要调用该方法，线程必须获得内部的对象锁。

换句话说，

```
public synchronized void method()
{
   method body
}
```

等价于

```
public void method()
{
   this.intrinsicLock.lock();
   try
   {
      method body
   }
```

```
      finally { this.intrinsicLock.unlock(); }
   }
```

例如，可以简单地声明Bank类的transfer方法为synchronized，而不是使用一个显式的锁。

内部对象锁只有一个相关条件。wait方法添加一个线程到等待集中，notifyAll /notify方法解除等待线程的阻塞状态。换句话说，调用wait或notifyAll等价于

```
intrinsicCondition.await();
intrinsicCondition.signalAll();
```

注释：*wait、notifyAll以及notify方法是Object类的final方法。Condition方法必须被命名为await、signalAll和signal以便它们不会与那些方法发生冲突。*

例如，可以用Java实现Bank类如下：

```
class Bank
{
   public synchronized void transfer(int from, int to, int amount) throws InterruptedException
   {
      while (accounts[from] < amount)
         wait(); // wait on intrinsic object lock's single condition
      accounts[from] -= amount;
      accounts[to] += amount;
      notifyAll(); // notify all threads waiting on the condition
   }
   public synchronized double getTotalBalance() { . . . }
   private double[] accounts;
}
```

可以看到，使用synchronized关键字来编写代码要简洁得多。当然，要理解这一代码，你必须了解每一个对象有一个内部锁，并且该锁有一个内部条件。由锁来管理那些试图进入synchronized方法的线程，由条件来管理那些调用wait的线程。

提示：*Synchronized方法是相对简单的。但是，初学者常常对条件感到困惑。在使用wait/notifyAll之前，应该考虑使用第14.10节描述的结构之一。*

将静态方法声明为synchronized也是合法的。如果调用这种方法，该方法获得相关的类对象的内部锁。例如，如果Bank类有一个静态同步的方法，那么当该方法被调用时，Bank.class对象的锁被锁住。因此，没有其他线程可以调用同一个类的这个或任何其他的同步静态方法。

内部锁和条件存在一些局限。包括：

- 不能中断一个正在试图获得锁的线程。
- 试图获得锁时不能设定超时。
- 每个锁仅有单一的条件，可能是不够的。

在代码中应该使用哪一种？Lock和Condition对象还是同步方法?下面是一些建议：

- 最好既不使用Lock/Condition也不使用synchronized关键字。在许多情况下你可以使用java.util.concurrent包中的一种机制，它会为你处理所有的加锁。例如，在第14.6节，你会看到如何使用阻塞队列来同步完成一个共同任务的线程。

- 如果synchronized关键字适合你的程序，那么请尽量使用它，这样可以减少编写的代码数量，减少出错的几率。例14-9给出了用同步方法实现的银行实例。
- 如果特别需要Lock/Condition结构提供的独有特性时，才使用Lock/Condition。

例14-9 Bank.java

```java
/**
 * A bank with a number of bank accounts that uses synchronization primitives.
 * @version 1.30 2004-08-01
 * @author Cay Horstmann
 */
public class Bank
{
   /**
    * Constructs the bank.
    * @param n the number of accounts
    * @param initialBalance the initial balance for each account
    */
   public Bank(int n, double initialBalance)
   {
      accounts = new double[n];
      for (int i = 0; i < accounts.length; i++)
         accounts[i] = initialBalance;
   }

   /**
    * Transfers money from one account to another.
    * @param from the account to transfer from
    * @param to the account to transfer to
    * @param amount the amount to transfer
    */
   public synchronized void transfer(int from, int to, double amount) throws InterruptedException
   {
      while (accounts[from] < amount)
         wait();
      System.out.print(Thread.currentThread());
      accounts[from] -= amount;
      System.out.printf(" %10.2f from %d to %d", amount, from, to);
      accounts[to] += amount;
      System.out.printf(" Total Balance: %10.2f%n", getTotalBalance());
      notifyAll();
   }

   /**
    * Gets the sum of all account balances.
    * @return the total balance
    */
   public synchronized double getTotalBalance()
   {
      double sum = 0;

      for (double a : accounts)
         sum += a;

      return sum;
```

```
    }

    /**
     * Gets the number of accounts in the bank.
     * @return the number of accounts
     */
    public int size()
    {
       return accounts.length;
    }

    private final double[] accounts;
}
```

API java.lang.Object 1.0

- `void notifyAll()`

  解除那些在该对象上调用wait方法的线程的阻塞状态。该方法只能在同步方法或同步块内部调用。如果当前线程不是对象锁的持有者，该方法抛出一个IllegalMonitorStateException异常。

- `void notify()`

  随机选择一个在该对象上调用wait方法的线程，解除其阻塞状态。该方法只能在一个同步方法或同步块中调用。如果当前线程不是对象锁的持有者，该方法抛出一个IllegalMonitorStateException异常。

- `void wait()`

  导致线程进入等待状态直到它被通知。该方法只能在一个同步方法中调用。如果当前线程不是对象锁的持有者，该方法抛出一个IllegalMonitorStateException异常。

- `void wait(long millis)`
- `void wait(long millis, int nanos)`

  导致线程进入等待状态直到它被通知或者经过指定的时间。这些方法只能在一个同步方法中调用。如果当前线程不是对象锁的持有者该方法抛出一个IllegalMonitorStateException异常。

  参数：millis　　毫秒数

  　　　nanos　　纳秒数，<1 000 000

14.5.6 同步阻塞

正如刚刚讨论的，每一个Java对象有一个锁。线程可以通过调用同步方法获得锁。还有另一种机制可以获得锁，通过进入一个同步阻塞。当线程进入如下形式的阻塞：

```
synchronized (obj) // this is the syntax for a synchronized block
{
   critical section
}
```

于是它获得obj的锁。

有时会发现“特殊的”锁，例如：

```
public class Bank
{
   public void transfer(int from, int to, int amount)
   {
      synchronized (lock) // an ad-hoc lock
      {
         accounts[from] -= amount;
         accounts[to] += amount;
      }
      System.out.println(. . .);
   }
   . . .
   private double[] accounts;
   private Object lock = new Object();
}
```

在此，lock对象被创建仅仅是用来使用每个Java对象持有的锁。

有时程序员使用一个对象的锁来实现额外的原子操作，实际上称为*客户端锁定*（client-side locking）。例如，考虑Vector类，一个列表，它的方法是同步的。现在，假定在Vector<Double>中存储银行余额。这里有一个transfer方法的原始实现：

```
public void transfer(Vector<Double> accounts, int from, int to, int amount) // ERROR
{
   accounts.set(from, accounts.get(from) - amount);
   accounts.set(to, accounts.get(to) + amount);
   System.out.println(. . .);
}
```

Vector类的get和set方法是同步的，但是，这对于我们并没有什么帮助。在第一次对get的调用已经完成之后，一个线程完全可能在transfer方法中被剥夺运行权。于是，另一个线程可能在相同的存储位置存入不同的值。但是，我们可以截获这个锁：

```
public void transfer(Vector<Double> accounts, int from, int to, int amount)
{
   synchronized (accounts)
   {
      accounts.set(from, accounts.get(from) - amount);
      accounts.set(to, accounts.get(to) + amount);
   }
   System.out.println(. . .);
}
```

这个方法可以工作，但是它完全依赖于这样一个事实，Vector类对自己的所有可修改方法都使用内部锁。然而，这是真的吗？Vector类的文档没有给出这样的承诺。不得不仔细研究源代码并希望将来的版本能介绍非同步的可修改方法。如你所见，客户端锁定是非常脆弱的，通常不推荐使用。

14.5.7 监视器概念

锁和条件是线程同步的强大工具，但是，严格地讲，它们不是面向对象的。多年来，研究

人员努力寻找一种方法，可以在不需要程序员考虑如何加锁的情况下，就可以保证多线程的安全性。最成功的解决方案之一是监视器（monitor），这一概念最早是由Per Brinch Hansen和Tony Hoare 在20世纪70年代提出的。用Java的术语来讲，监视器具有如下特性：

- 监视器是只包含私有域的类。
- 每个监视器类的对象有一个相关的锁。
- 使用该锁对所有的方法进行加锁。换句话说，如果客户端调用obj.method()，那么obj对象的锁是在方法调用开始时自动获得，并且当方法返回时自动释放该锁。因为所有的域是私有的，这样的安排可以确保一个线程在对对象操作时，没有其他线程能访问该域。
- 该锁可以有任意多个相关条件。

监视器的早期版本只有单一的条件，使用一种很优雅的句法。可以简单地调用await accounts[from] >= balance而不使用任何显式的条件变量。然而，研究表明盲目地重新测试条件是低效的。显式的条件变量解决了这一问题。每一个条件变量管理一个独立的线程集。

Java设计者以不是很精确的方式采用了监视器概念，Java中的每一个对象有一个内部的锁和内部的条件。如果一个方法用synchronized关键字声明，那么，它表现的就像是一个监视器方法。通过调用wait/notifyAll/notify来访问条件变量。

然而，在下述的3个方面Java对象不同于监视器，从而使得线程的安全性下降：

- 域不要求必须是private。
- 方法不要求必须是synchronized。
- 内部锁对客户是可用的。

这种对安全性的轻视激怒了Per Brinch Hansen。他在一次对原始Java中的多线程的严厉评论中，写到："这实在是令我震惊，在监视器和并发Pascal出现四分之一个世纪后，Java的这种不安全的并行机制被编程社区接受。这没有任何益处。" [Java's Insecure Parallelism, ACM SIGPLAN Notices 34:38-45, April 1999.]

14.5.8 Volatile域

有时，仅仅为了读写一个或两个实例域就使用同步，显得开销过大了。毕竟，什么地方能出错呢？遗憾的是，使用现代的处理器与编译器，出错的可能性很大。

- 多处理器的计算机能够暂时在寄存器或本地内存缓冲区中保存内存中的值。结果是，运行在不同处理器上的线程可能在同一个内存位置取到不同的值。
- 编译器可以改变指令执行的顺序以使吞吐量最大化。这种顺序上的变化不会改变代码语义，但是编译器假定内存的值仅仅在代码中有显式的修改指令时才会改变。然而，内存的值可以被另一个线程改变！

如果你使用锁来保护可以被多个线程访问的代码，那么可以不考虑这种问题。编译器被要求通过在必要的时候刷新本地缓存来保持锁的效应，并且不能不正当地重新排序指令。详细的解释见JSR 133的Java内存模型和线程规范（参看 http://www.jcp.org/en/jsr/detail?id=133）。该规范的大部分很复杂而且技术性强，但是文档中也包含了很多解释得很清晰的例子。在http://www-106.ibm.com/developerworks/java/ library/j-jtp02244.html有Brian Goetz写的一个更

易懂的概要介绍。

注释：Brian Goetz给出了下述“同步格言”：“如果向一个变量写入值，而这个变量接下来可能会被另一个线程读取，或者，从一个变量读值，而这个变量可能是之前被另一个线程写入的，此时必须使用同步”。

volatile关键字为实例域的同步访问提供了一种免锁机制。如果声明一个域为volatile，那么编译器和虚拟机就知道该域是可能被另一个线程并发更新的。

例如，假定一个对象有一个布尔标记done，它的值被一个线程设置却被另一个线程查询，如同我们讨论过的那样，你可以使用锁：

```
public synchronized boolean isDone() { return done; }
public synchronized void setDone() { done = true; }
private boolean done;
```

或许使用内部锁不是个好主意。如果另一个线程已经对该对象加锁，isDone和setDone方法可能阻塞。如果注意到这个方面，一个线程可以为这一变量使用独立的Lock。但是，这也会带来许多麻烦。

在这种情况下，将域声明为volatile是合理的：

```
public boolean isDone() { return done; }
public void setDone() { done = true; }
private volatile boolean done;
```

警告：Volatile变量不能提供原子性。例如，方法

```
public void flipDone() { done = !done; } // not atomic
```

不能确保改变域中的值。

在这样一种非常简单的情况下，存在第3种可能性，使用 AtomicBoolean。这个类有方法get和set，且确保是原子的（就像它们是同步的一样）。该实现使用有效的机器指令，在不使用锁的情况下确保原子性。在java.util.concurrent.atomic中有许多包装器类用于原子的整数、浮点数、数组等。这些类是为编写并发实用程序的系统程序员提供使用的，而不是应用程序员。

总之，在以下3个条件下，域的并发访问是安全的：

- 域是final，并且在构造器调用完成之后被访问。
- 对域的访问由公有的锁进行保护。
- 域是volatile的。

注释：Java SE 5.0之前，volatile的语义是允许的。语言的设计者试图在优化使用volatile域的代码的性能方面给实现人员留有余地。但是，旧规范太复杂，实现人员难以理解，这带来了混乱和非预期的行为。例如，不可变对象不是真的不可变。

14.5.9 死锁

锁和条件不能解决多线程中的所有问题。考虑下面的情况：

```
账户1：$200
```

```
账户2：$300
线程1：从账户1转移$300到账户2
线程2：从账户2转移$400到账户1
```

如图14-6所示，线程1和线程2都被阻塞了。因为账户1以及账户2中的余额都不足以进行转账，两个线程都无法执行下去。

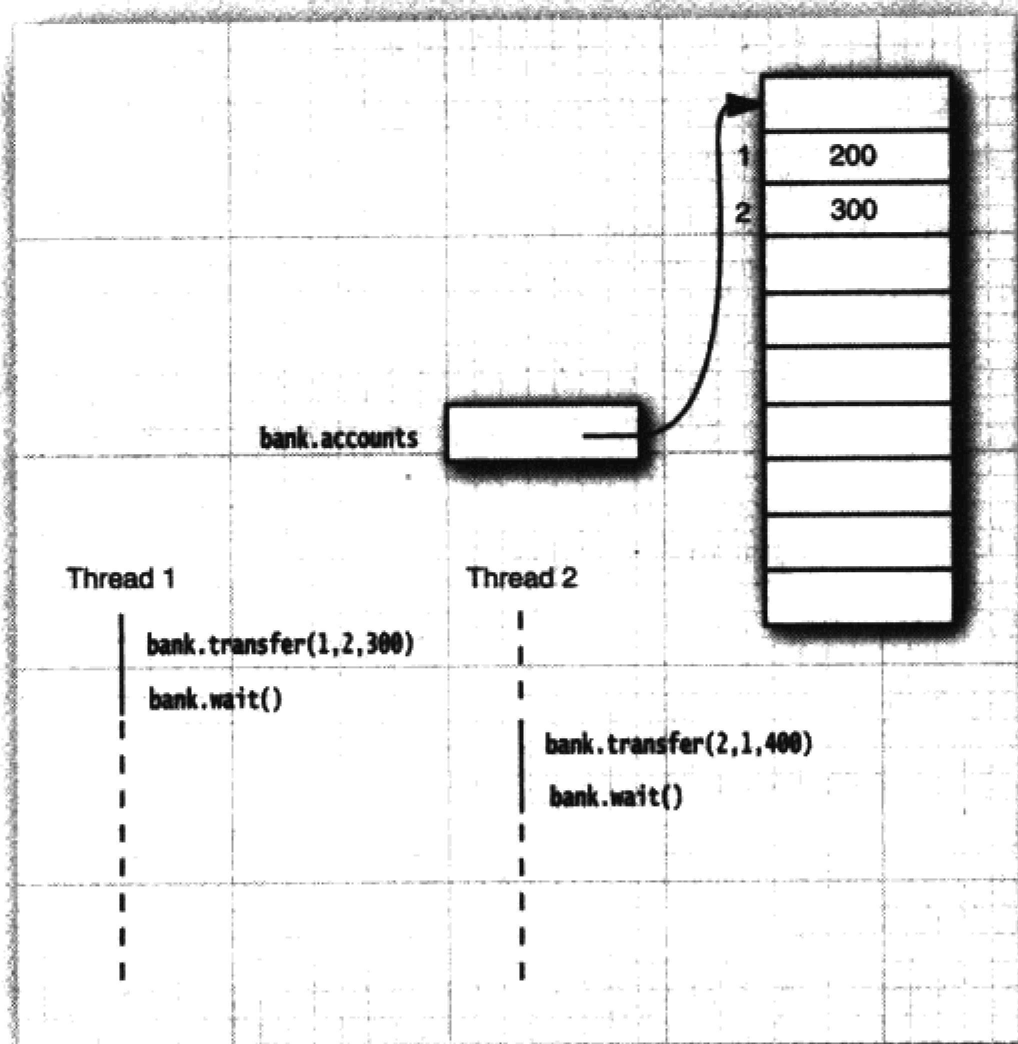


图14-6 发生死锁的情况

是否会因为每一个线程要等待更多的钱款存入而导致所有线程都被阻塞呢？这样的状态称为死锁（deadlock）。

在这个程序里，死锁不会发生，原因很简单。每一次转账至多$1 000。因为有100个账户，而且所有账户的总金额是$100 000，在任意时刻，至少有一个账户的余额高于$1 000。从该账户取钱的线程可以继续运行。

但是，如果修改run方法，把每次转账至多$1 000的限制去掉，死锁很快就会发生。试试看。将NACCOUNTS设为10。每次交易的金额上限设置为2 * INITIAL_BALANCE，然后运行该程序。程序将运行一段时间后就会挂起。

> **提示**：当程序挂起时，键入CTRL+\，将得到一个所有线程的列表。每一个线程有一个栈踪迹，告诉你线程被阻塞的位置。像第11章叙述的那样，运行jconsole并参考线程面板（见图14-7）。

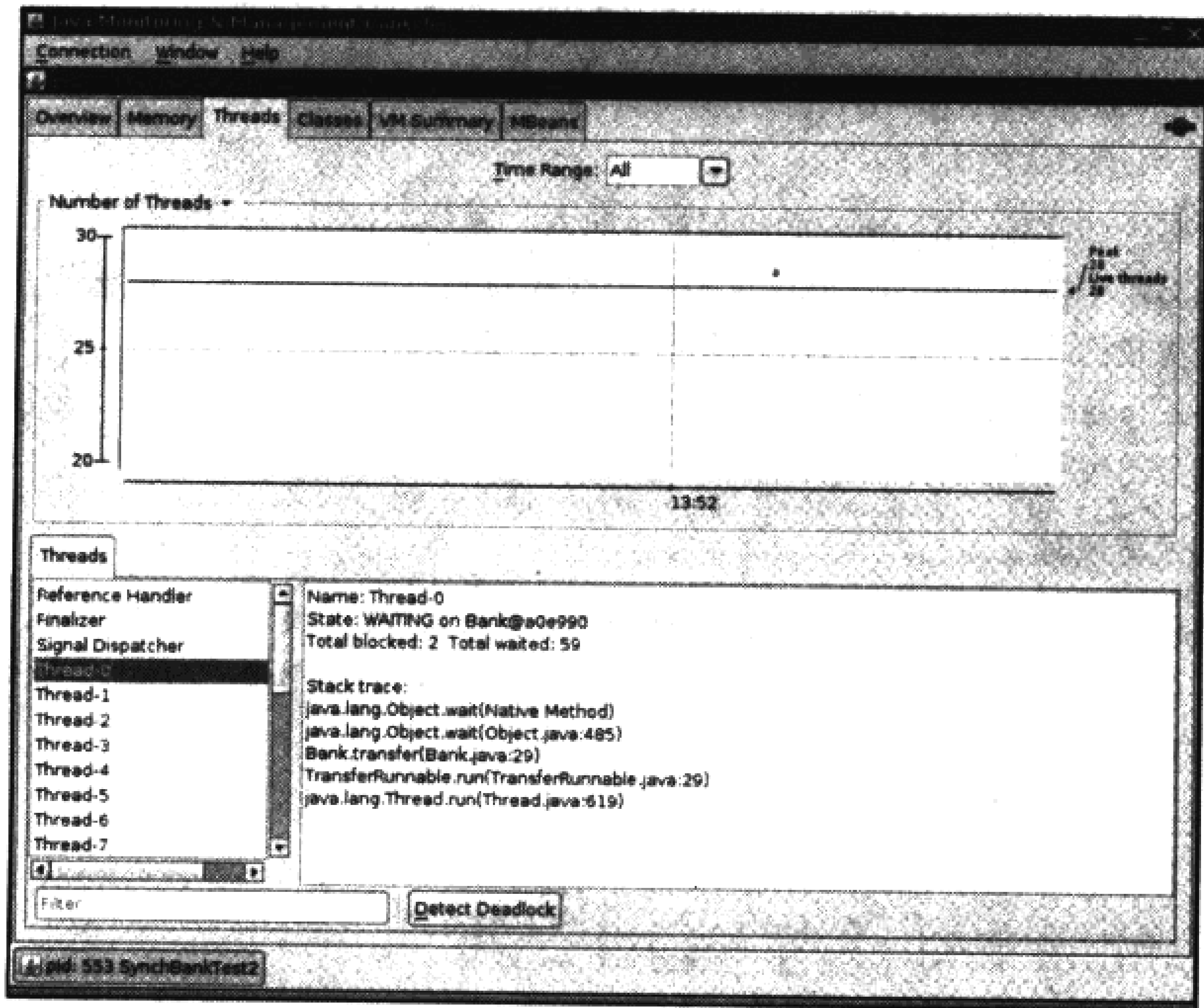


图14-7 jconsole中的线程面板

导致死锁的另一种途径是让第i个线程负责向第i个账户存钱，而不是从第i个账户取钱。这样一来，有可能将所有的线程都集中到一个账户上，每一个线程都试图从这个账户中取出大于该账户余额的钱。试试看。在SynchBankTest程序中，转用TransferRunnable类的run方法。在调用transfer时，交换fromAccount和toAccount。运行该程序并查看它为什么会立即死锁。

还有一种很容易导致死锁的情况：在SynchBankTest程序中，将signalAll方法转换为signal，会发现该程序最终会挂起（将NACCOUNTS设为10可以更快地看到结果）。signalAll通知所有等待增加资金的线程，与此不同的是signal方法仅仅对一个线程解锁。如果该线程不能继续运行，所有的线程可能都被阻塞。考虑下面这个会发生死锁的例子。

账户1：$1 990
所有其他账户：每一个$990
线程1：从账户1转移$995到账户2
所有其他线程：从他们的账户转移$995到另一个账户

显然，除了线程1，所有的线程都被阻塞，因为他们的账户中没有足够的余额。

线程1继续执行，运行后出现如下状况：

账户1：$995
账户2：$1 985
所有其他账户：每个$990

然后，线程1调用signal。signal方法随机选择一个线程为它解锁。假定它选择了线程3。该

线程被唤醒，发现在它的账户里没有足够的金额，它再次调用await。但是，线程1仍在运行，将随机地产生一个新的交易，例如，

线程1：从账户1转移$997到账户2

现在，线程1也调用await，所有的线程都被阻塞。系统死锁。

问题的起因在于调用 signal。它仅仅为一个线程解锁，而且，它很可能选择一个不能继续运行的线程（在我们的例子中，线程2必须把钱从账户2中取出）。

遗憾的是，Java编程语言中没有任何东西可以避免或打破这种死锁现象。必须仔细设计程序，以确保不会出现死锁。

14.5.10 锁测试与超时

线程在调用lock方法来获得另一个线程所持有的锁的时候，很可能发生阻塞。应该更加谨慎地申请锁。tryLock方法试图申请一个锁，在成功获得锁后返回true，否则，立即返回false，而且线程可以立即离开去做其他事情。

```
if (myLock.tryLock())
   // now the thread owns the lock
   try { . . . }
   finally { myLock.unlock(); }
else
  // do something else
```

可以调用tryLock时，使用超时参数，像这样：

```
if (myLock.tryLock(100, TimeUnit.MILLISECONDS)) . . .
```

TimeUnit是一个枚举类型，可以取的值包括SECONDS、MILLISECONDS、MICROSECONDS和NANOSECONDS。

lock方法不能被中断。如果一个线程在等待获得一个锁时被中断，中断线程在获得锁之前一直处于阻塞状态。如果出现死锁，那么，lock方法就无法终止。

然而，如果调用带有用超时参数的tryLock，那么如果线程在等待期间被中断，将抛出InterruptedException异常。这是一个非常有用的特性，因为允许程序打破死锁。

也可以调用lockInterruptibly方法。它就相当于一个超时设为无限的tryLock方法。

在等待一个条件时，也可以提供一个超时：

```
myCondition.await(100, TimeUnit.MILLISECONDS))
```

如果一个线程被另一个线程通过调用signalAll或 signal激活，或者超时时限已达到，或者线程被中断，那么await方法将返回。

如果等待的线程被中断，await方法将抛出一个InterruptedException异常。在你希望出现这种情况时线程继续等待（可能不太合理），可以使用awaitUninterruptibly方法代替await。

API java.util.concurrent.locks.Lock 5.0

• `boolean tryLock()`

尝试获得锁而没有发生阻塞；如果成功返回真。这个方法会抢夺可用的锁，即使该锁有

公平加锁策略，即便其他线程已经等待很久也是如此。

- `boolean tryLock(long time, TimeUnit unit)`

  尝试获得锁，阻塞时间不会超过给定的值；如果成功返回true。

- `void lockInterruptibly()`

  获得锁，但是会不确定地发生阻塞。如果线程被中断，抛出一个InterruptedException异常。

API `java.util.concurrent.locks.Condition` 5.0

- `boolean await(long time, TimeUnit unit)`

  进入该条件的等待集，直到线程从等待集中移出或等待了指定的时间之后才解除阻塞。如果因为等待时间到了而返回就返回false，否则返回true。

- `void awaitUninterruptibly()`

  进入该条件的等待集，直到线程从等待集移出才解除阻塞。如果线程被中断，该方法不会抛出InterruptedException异常。

14.5.11 读/写锁

java.util.concurrent.locks包定义了两个锁类，我们已经讨论的ReentrantLock类和ReentrantReadWriteLock类。如果很多线程从一个数据结构读取数据而很少线程修改其中数据的话，后者是十分有用的。在这种情况下，允许对读者线程共享访问是合适的。当然，写者线程依然必须是互斥访问的。

下面是使用读/写锁的必要步骤：

1）构造一个ReentrantReadWriteLock对象：

```
private ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
```

2）抽取读锁和写锁：

```
private Lock readLock = rwl.readLock();
private Lock writeLock = rwl.writeLock();
```

3）对所有的访问者加读锁：

```
public double getTotalBalance()
{
   readLock.lock();
   try { . . . }
   finally { readLock.unlock(); }
}
```

4）对所有的修改者加写锁：

```
public void transfer(. . .)
{
   writeLock.lock();
   try { . . . }
   finally { writeLock.unlock(); }
}
```

java.util.concurrent.locks.ReentrantReadWriteLock 5.0

- `Lock readLock()`

  得到一个可以被多个读操作共用的读锁，但会排斥所有写操作。

- `Lock writeLock()`

  得到一个写锁，排斥所有其他的读操作和写操作。

14.5.12 为什么弃用 stop 和 suspend 方法

初始的Java版本定义了一个stop方法用来终止一个线程，以及一个suspend方法用来阻塞一个线程直至另一个线程调用resume。stop和suspend方法有一些共同点：都试图控制一个给定线程的行为。

从Java SE 1.2起就弃用了这两个方法。stop方法天生就不安全，经验证明suspend方法会经常导致死锁。在本节，将看到这些方法的问题所在，以及怎样避免这些问题的出现。

首先来看看stop方法，该方法终止所有未结束的方法，包括run方法。当线程被终止，立即释放被它锁住的所有对象的锁。这会导致对象处于不一致的状态。例如，假定TransferThread在从一个账户向另一个账户转账的过程中被终止，钱款已经转出，却没有转入目标账户，现在银行对象就被破坏了。因为锁已经被释放，这种破坏会被其他尚未停止的线程观察到。

当线程要终止另一个线程时，无法知道什么时候调用stop方法是安全的，什么时候导致对象被破坏。因此，该方法被弃用了。在希望停止线程的时候应该中断线程，被中断的线程会在安全的时候停止。

注释：一些作者声称stop方法被弃用是因为它会导致对象被一个已停止的线程永久锁定。但是，这一说法是错误的。从技术上讲，被停止的线程通过抛出ThreadDeath异常退出所有它所调用的同步方法。结果是，该线程释放它持有的内部对象锁。

接下来，看看suspend方法有什么问题。与stop不同，suspend不会破坏对象。但是，如果用suspend挂起一个持有一个锁的线程，那么，该锁在恢复之前是不可用的。如果调用suspend方法的线程试图获得同一个锁，那么程序死锁：被挂起的线程等着被恢复，而将其挂起的线程等待获得锁。

在图形用户界面中经常出现这种情况。假定我们有一个图形化的银行模拟程序。Pause按钮用来挂起转账线程，而Resume按钮用来恢复线程。

```
pauseButton.addActionListener(new
   ActionListener()
   {
      public void actionPerformed(ActionEvent event)
      {
         for (int i = 0; i < threads.length; i++)
            threads[i].suspend(); // Don't do this
      }
   });
resumeButton.addActionListener(. . .); // calls resume on all transfer threads
```

假设有一个paintComponent方法，通过调用getBalances方法获得一个余额数组，从而为每

一个账户绘制图表。

就像在第14.11节所看到的，按钮动作和重绘动作出现在同一个线程中——事件分配线程（event dispatch thread）。考虑下面的情况：

1）某个转账线程获得bank对象的锁。

2）用户点击Pause按钮。

3）所有转账线程被挂起；其中之一仍然持有bank对象上的锁。

4）因为某种原因，该账户图表需要重新绘制。

5）paintComponent方法调用getBalances方法。

6）该方法试图获得bank对象的锁。

现在程序被冻结了。

事件分配线程不能继续运行，因为锁由一个被挂起的线程所持有。因此，用户不能点击Resume按钮，并且这些线程无法恢复。

如果想安全地挂起线程，引入一个变量suspendRequested并在run方法的某个安全的地方测试它，安全的地方是指该线程没有封锁其他线程需要的对象的地方。当该线程发现suspendRequested变量已经设置，将会保持等待状态直到它再次获得为止。

下面的代码框架实现这一设计：

```
public void run()
{
   while (. . .)
   {
      . . .
      if (suspendRequested)
      {
         suspendLock.lock();
         try { while (suspendRequested) suspendCondition.await(); }
         finally { suspendLock.unlock(); }
      }
   }
}
public void requestSuspend() { suspendRequested = true; }
public void requestResume()
{
   suspendRequested = false;
   suspendLock.lock();
   try { suspendCondition.signalAll(); }
   finally { suspendLock.unlock(); }
}
private volatile boolean suspendRequested = false;
private Lock suspendLock = new ReentrantLock();
private Condition suspendCondition = suspendLock.newCondition();
```

14.6 阻塞队列

现在，读者已经看到了形成Java并发程序设计基础的底层构建块。然而，对于实际编程来说，应该尽可能远离底层结构。使用由并发处理的专业人士实现的较高层次的结构要方便得多、要安全得多。

对于许多线程问题，可以通过使用一个或多个队列以优雅且安全的方式将其形式化。生产者线程向队列插入元素，消费者线程则取出它们。使用队列，可以安全地从一个线程向另一个线程传递数据。例如，考虑银行转账程序，转账线程将转账指令对象插入一个队列中，而不是直接访问银行对象。另一个线程从队列中取出指令执行转账。只有该线程可以访问该银行对象的内部。因此不需要同步。（当然，线程安全的队列类的实现者不能不考虑锁和条件。但是，那是他们的问题而不是你的问题。）

当试图向队列添加元素而队列已满，或是想从队列移出元素而队列为空的时候，阻塞队列（blocking queue）导致线程阻塞。在协调多个线程之间的合作时，阻塞队列是一个有用的工具。工作者线程可以周期性地将中间结果存储在阻塞队列中。其他的工作者线程移出中间结果并进一步加以修改。队列会自动地平衡负载。如果第一个线程集运行得比第二个慢，第二个线程集在等待结果时会阻塞。如果第一个线程集运行得快，它将等待第二个队列集赶上来。表14-1给出了阻塞队列的方法。

表14-1 阻塞队列方法

方法	正常动作	特殊情况下的动作
add	添加一个元素	如果队列满，则抛出IllegalStateException异常
element	返回队列的头元素	如果队列空，抛出NoSuchElementException异常
offer	添加一个元素并返回true	如果队列满,返回false
peek	返回队列的头元素	如果队列空，则返回null
poll	移出并返回队列的头元素	如果队列空，则返回null
put	添加一个元素	如果队列满，则阻塞
remove	移出并返回头元素	如果队列空，则抛出NoSuchElementException异常
take	移出并返回头元素	如果队列空，则阻塞

阻塞队列方法分为以下3类，这取决于当队列满或空时它们的响应方式。如果将队列当作线程管理工具来使用，将要用到put和take方法。当试图向满的队列中添加或从空的队列中移出元素时，add、remove和element操作抛出异常。当然，在一个多线程程序中，队列会在任何时候空或满，因此，一定要使用offer、poll和peek方法作为替代。这些方法如果不能完成任务，只是给出一个错误提示而不会抛出异常。

✔ **注释**：poll和peek方法返回空来指示失败。因此，向这些队列中插入null值是非法的。

还有带有超时的offer方法和poll方法的变体。例如，下面的调用：

```
boolean success = q.offer(x, 100, TimeUnit.MILLISECONDS);
```

尝试在100毫秒的时间内在队列的尾部插入一个元素。如果成功返回true；否则，达到超时时，返回false。类似地，下面的调用：

```
Object head = q.poll(100, TimeUnit.MILLISECONDS)
```

尝试用100毫秒的时间移除队列的头元素；如果成功返回头元素，否则，达到在超时时，返回null。

如果队列满，则put方法阻塞；如果队列空，则take方法阻塞。在不带超时参数时，offer和

poll方法等效。

java.util.concurrent包提供了阻塞队列的几个变种。默认情况下，LinkedBlockingQueue的容量是没有上边界的，但是，也可以选择指定最大容量。LinkedBlockingDeque是一个双端的版本。ArrayBlockingQueue在构造时需要指定容量，并且有一个可选的参数来指定是否需要公平性。若设置了公平参数，则那么等待了最长时间的线程会优先得到处理。通常，公平性会降低性能，只有在确实非常需要时才使用它。

PriorityBlockingQueue是一个带优先级的队列，而不是先进先出队列。元素按照它们的优先级顺序被移出。该队列是没有容量上限，但是，如果队列是空的，取元素的操作会阻塞。（有关优先级队列的详细内容参看第13章。）

最后，DelayQueue包含实现Delayed接口的对象：

```
interface Delayed extends Comparable<Delayed>
{
   long getDelay(TimeUnit unit);
}
```

getDelay方法返回对象的残留延迟。负值表示延迟已经结束。元素只有在延迟用完的情况下才能从DelayQueue移除。还必须实现compareTo方法。DelayQueue使用该方法对元素进行排序。

例14-10中的程序展示了如何使用阻塞队列来控制线程集。程序在一个目录及它的所有子目录下搜索所有文件，打印出包含指定关键字的行。

例14-10 BlockingQueueTest.java

```
import java.io.*;
import java.util.*;
import java.util.concurrent.*;

/**
 * @version 1.0 2004-08-01
 * @author Cay Horstmann
 */
public class BlockingQueueTest
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);
      System.out.print("Enter base directory (e.g. /usr/local/jdk1.6.0/src): ");
      String directory = in.nextLine();
      System.out.print("Enter keyword (e.g. volatile): ");
      String keyword = in.nextLine();

      final int FILE_QUEUE_SIZE = 10;
      final int SEARCH_THREADS = 100;

      BlockingQueue<File> queue = new ArrayBlockingQueue<File>(FILE_QUEUE_SIZE);

      FileEnumerationTask enumerator = new FileEnumerationTask(queue, new File(directory));
      new Thread(enumerator).start();
      for (int i = 1; i <= SEARCH_THREADS; i++)
         new Thread(new SearchTask(queue, keyword)).start();
   }
```

```
}

/**
 * This task enumerates all files in a directory and its subdirectories.
 */
class FileEnumerationTask implements Runnable
{
   /**
    * Constructs a FileEnumerationTask.
    * @param queue the blocking queue to which the enumerated files are added
    * @param startingDirectory the directory in which to start the enumeration
    */
   public FileEnumerationTask(BlockingQueue<File> queue, File startingDirectory)
   {
      this.queue = queue;
      this.startingDirectory = startingDirectory;
   }

   public void run()
   {
      try
      {
         enumerate(startingDirectory);
         queue.put(DUMMY);
      }
      catch (InterruptedException e)
      {
      }
   }

   /**
    * Recursively enumerates all files in a given directory and its subdirectories
    * @param directory the directory in which to start
    */
   public void enumerate(File directory) throws InterruptedException
   {
      File[] files = directory.listFiles();
      for (File file : files)
      {
         if (file.isDirectory()) enumerate(file);
         else queue.put(file);
      }
   }

   public static File DUMMY = new File("");

   private BlockingQueue<File> queue;
   private File startingDirectory;
}

/**
 * This task searches files for a given keyword.
 */
class SearchTask implements Runnable
{
   /**
    * Constructs a SearchTask.
```

```
    * @param queue the queue from which to take files
    * @param keyword the keyword to look for
    */
   public SearchTask(BlockingQueue<File> queue, String keyword)
   {
      this.queue = queue;
      this.keyword = keyword;
   }

   public void run()
   {
      try
      {
         boolean done = false;
         while (!done)
         {
            File file = queue.take();
            if (file == FileEnumerationTask.DUMMY)
            {
               queue.put(file);
               done = true;
            }
            else search(file);
         }
      }
      catch (IOException e)
      {
         e.printStackTrace();
      }
      catch (InterruptedException e)
      {
      }
   }

   /**
    * Searches a file for a given keyword and prints all matching lines.
    * @param file the file to search
    */
   public void search(File file) throws IOException
   {
      Scanner in = new Scanner(new FileInputStream(file));
      int lineNumber = 0;
      while (in.hasNextLine())
      {
         lineNumber++;
         String line = in.nextLine();
         if (line.contains(keyword)) System.out.printf("%s:%d:%s%n", file.getPath(),
               lineNumber, line);
      }
      in.close();
   }

   private BlockingQueue<File> queue;
   private String keyword;
}
```

生产者线程枚举在所有子目录下的所有文件并把它们放到一个阻塞队列中。这个操作很快，如果没有上限的话，很快就包含了所有找到的文件。

我们同时启动了大量搜索线程。每个搜索线程从队列中取出一个文件，打开它，打印所有包含该关键字的行，然后取出下一个文件。我们使用一个小技巧在工作结束后终止这个应用程序。为了发出完成信号，枚举线程放置一个虚拟对象到队列中（这就像在行李输送带上放一个写着“最后一个包”的虚拟包）。当搜索线程取到这个虚拟对象时，将其放回并终止。

注意，不需要显式的线程同步。在这个应用程序中，我们使用队列数据结构作为一种同步机制。

API java.util.concurrent.ArrayBlockingQueue<E> 5.0

- `ArrayBlockingQueue(int capacity)`
- `ArrayBlockingQueue(int capacity, boolean fair)`

  构造一个带有指定的容量和公平性设置的阻塞队列。该队列用循环数组实现。

API java.util.concurrent.LinkedBlockingQueue<E> 5.0

API java.util.concurrent.LinkedBlockingDeque<E> 6

- `LinkedBlockingQueue()`
- `LinkedBlockingDeque()`

  构造一个无上限的阻塞队列或双向队列，用链表实现。
- `LinkedBlockingQueue(int capacity)`
- `LinkedBlockingDeque(int capacity)`

  根据指定容量构建一个有限的阻塞队列或双向队列，用链表实现。

API java.util.concurrent.DelayQueue<E extends Delayed> 5.0

- `DelayQueue()`

  构造一个包含Delayed元素的无界的阻塞时间有限的阻塞队列。只有那些延迟已经超过时间的元素可以从队列中移出。

API java.util.concurrent.Delayed 5.0

- `long getDelay(TimeUnit unit)`

  得到该对象的延迟，用给定的时间单位进行度量。

API java.util.concurrent.PriorityBlockingQueue<E> 5.0

- `PriorityBlockingQueue()`
- `PriorityBlockingQueue(int initialCapacity)`
- `PriorityBlockingQueue(int initialCapacity, Comparator<? super E> comparator)`

构造一个无边界阻塞优先队列，用堆实现。

参数：initialCapacity　　优先队列的初始容量。默认值是11。

comparator　　用来对元素进行比较的比较器，如果没有指定，则元素必须实现Comparable接口。

API java.util.concurrent.BlockingQueue<E> 5.0

- `void put(E element)`
  添加元素，在必要时阻塞。
- `E take()`
  移除并返回头元素，必要时阻塞。
- `boolean offer(E element, long time, TimeUnit unit)`
  添加给定的元素，如果成功返回true，如果必要时阻塞，直至元素已经被添加或超时。
- `E poll(long time, TimeUnit unit)`
  移除并返回头元素，必要时阻塞，直至元素可用或超时用完。失败时返回null。

API java.util.concurrent.BlockingDeque<E> 6

- `void putFirst(E element)`
- `void putLast(E element)`
  添加元素，必要时阻塞。
- `E takeFirst()`
- `E takeLast()`
  移除并返回头元素或尾元素，必要时阻塞。
- `boolean offerFirst(E element, long time, TimeUnit unit)`
- `boolean offerLast(E element, long time, TimeUnit unit)`
  添加给定的元素，成功时返回true，必要时阻塞直至元素被添加或超时。
- `E pollFirst(long time, TimeUnit unit)`
- `E pollLast(long time, TimeUnit unit)`
  移动并返回头元素或尾元素，必要时阻塞，直至元素可用或超时。失败时返回空。

14.7 线程安全的集合

如果多线程要并发地修改一个数据结构，例如散列表，那么很容易会破坏这个数据结构（有关散列表的详细信息见第13章）。例如，一个线程可能要开始向表中插入一个新元素。假定在调整散列表各个桶之间的链接关系的过程中，被剥夺了控制权。如果另一个线程也开始遍历同一个链表，可能使用无效的链接并造成混乱，会抛出异常或者陷入死循环。

可以通过提供锁来保护共享数据结构，但是选择线程安全的实现作为替代可能更容易些。当然，前一节讨论的阻塞队列就是线程安全的集合。在下面各小节中，将讨论Java类库提供的另一种线程安全的集合。

14.7.1 高效的映像、集合和队列

java.util.concurrent包提供了映像、有序集和队列的高效实现：ConcurrentHashMap、ConcurrentSkipListMap、ConcurrentSkipListSet和ConcurrentLinkedQueue。

这些集合使用复杂的算法，通过允许并发地访问数据结构的不同部分来使竞争极小化。

与大多数集合不同，size方法不必在常量时间内操作。确定这样的集合当前的大小通常需要遍历。

集合返回弱一致性（weakly consisteut）的迭代器。这意味着迭代器不一定能反映出它们被构造之后的所有的修改，但是，它们不会将同一个值返回两次，也不会抛出ConcurrentModificationException异常。

> **注释**：与之形成对照的是，集合如果在迭代器构造之后发生改变，java.util包中的迭代器将抛出一个ConcurrentModificationException异常。

并发的散列映像表，可高效地支持大量的读者和一定数量的写者。默认情况下，假定可以有多达16个写者线程同时执行。可以有更多的写者线程，但是，如果同一时间多于16个，其他线程将暂时被阻塞。可以指定更大数目的构造器，然而，恐怕没有这种必要。

ConcurrentHashMap和ConcurrentSkipListMap类有相应的方法用于原子性的关联插入以及关联删除。putIfAbsent方法自动地添加新的关联，前提是原来没有这一关联。对于多线程访问的缓存来说这是很有用的，确保只有一个线程向缓存添加项：

```
cache.putIfAbsent(key, value);
```

相反的操作是删除（或许应该叫作removeIfPresent）。调用

```
cache.remove(key, value)
```

将原子性地删除键值对，如果它们在映像表中出现的话。最后，

```
cache.replace(key, oldValue, newValue)
```

原子性地用新值替换旧值，假定旧值与指定的键值关联。

API java.util.concurrent.ConcurrentLinkedQueue<E> 5.0

- `ConcurrentLinkedQueue<E>()`

  构造一个可以被多线程安全访问的无边界非阻塞的队列。

API java.util.concurrent.ConcurrentLinkedQueue<E> 6

- `ConcurrentSkipListSet<E>()`
- `ConcurrentSkipListSet<E>(Comparator<? super E> comp)`

  构造一个可以被多线程安全访问的有序集。第一个构造器要求元素实现Comparable接口。

API java.util.concurrent.ConcurrentHashMap<K, V> 5.0

API java.util.concurrent.ConcurrentSkipListMap<K, V> 6

- `ConcurrentHashMap<K, V>()`

- `ConcurrentHashMap<K, V>(int initialCapacity)`
- `ConcurrentHashMap<K, V>(int initialCapacity, float loadFactor, int concurrencyLevel)`

  构造一个可以被多线程安全访问的散列映像表。

  参数：initialCapacity　集合的初始容量。默认值为16。

  　　　loadFactor　控制调整：如果每一个桶的平均负载超过这个因子，表的大小会被重新调整。默认值为0.75。

  　　　concurrencyLevel　并发写者线程的估计数目。
- `ConcurrentSkipListMap<K, V>()`
- `ConcurrentSkipListSet<K, V>(Comparator<? super K> comp)`

  构造一个可以被多线程安全访问的有序的映像表。第一个构造器要求键实现Comparable接口。
- `V putIfAbsent(K key, V value)`

  如果该键没有在映像表中出现，则将给定的值同给定的键关联起来，并返回null。否则返回与该键关联的现有值。
- `boolean remove(K key, V value)`

  如果给定的键与给定的值关联，删除给定的键与值并返回真。否则，返回false。
- `boolean replace(K key, V oldValue, V newValue)`

  如果给定的键当前与oldvalue相关联，用它与newValue关联。否则，返回false。

14.7.2 写数组的拷贝

CopyOnWriteArrayList和CopyOnWriteArraySet是线程安全的集合，其中所有的修改线程对底层数组进行复制。如果在集合上进行迭代的线程数超过修改线程数，这样的安排是很有用的。当构建一个迭代器的时候，它包含一个对当前数组的引用。如果数组后来被修改了，迭代器仍然引用旧数组，但是，集合的数组已经被替换了。因而，旧的迭代器拥有一致的（可能过时的）视图，访问它无须任何同步开销。

14.7.3 旧的线程安全的集合

从Java的初始版本开始，Vector和Hashtable类就提供了线程安全的动态数组和散列表的实现。在Java SE 1.2中，这些类被弃用了，取而代之的是ArrayList和HashMap类。这些类不是线程安全的，而集合库中提供了不同的机制。任何集合类通过使用**同步包装器**（synchronization wrapper）变成线程安全的：

```
List<E> synchArrayList = Collections.synchronizedList(new ArrayList<E>());
Map<K, V> synchHashMap = Collections.synchronizedMap(new HashMap<K, V>());
```

结果集合的方法使用锁加以保护，提供了线程的安全访问。

应该确保没有任何线程通过原始的非同步方法访问数据结构。最便利的方法是确保不保存任何指向原始对象的引用，简单地构造一个集合并立即传递给包装器，像我们的例子中所做的

那样。

如果在另一个线程可能进行修改时要对集合进行迭代，仍然需要使用“客户端”封锁：

```
synchronized (synchHashMap)
{
   Iterator<K> iter = synchHashMap.keySet().iterator();
   while (iter.hasNext()) . . .;
}
```

如果使用“for each”循环必须使用同样的代码，因为循环使用了迭代器。注意：如果在迭代过程中，别的线程修改集合，迭代器会失效，抛出ConcurrentModificationException异常。同步仍然是需要的，因此并发的修改可以被可靠地检测出来。

最好使用java.util.concurrent包中定义的集合，不使用同步包装器中的。特别是，假如它们访问的是不同的桶，由于ConcurrentHashMap已经精心地实现了，多线程可以访问它而且不会彼此阻塞。有一个例外是经常被修改的数组列表。 在那种情况下，同步的 ArrayList可以胜过CopyOnWriteArrayList。

API java.util.Collections 1.2

- `static <E> Collection<E> synchronizedCollection(Collection<E> c)`
- `static <E> List synchronizedList(List<E> c)`
- `static <E> Set synchronizedSet(Set<E> c)`
- `static <E> SortedSet synchronizedSortedSet(SortedSet<E> c)`
- `static <K, V> Map<K, V> synchronizedMap(Map<K, V> c)`
- `static <K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> c)`

  构建集合视图，该集合的方法是同步的。

14.8 Callable与Future

Runnable封装一个异步运行的任务，可以把它想像成为一个没有参数和返回值的异步方法。Callable与Runnable类似，但是有返回值。Callable接口是一个参数化的类型，只有一个方法call。

```
public interface Callable<V>
{
   V call() throws Exception;
}
```

类型参数是返回值的类型。例如，Callable<Integer> 表示一个最终返回Integer对象的异步计算。

Future保存异步计算的结果。可以启动一个计算，将Future对象交给某个线程，然后忘掉它。Future对象的所有者在结果计算好之后就可以获得它。

Future接口具有下面的方法：

```
public interface Future<V>
{
   V get() throws . . .;
   V get(long timeout, TimeUnit unit) throws . . .;
```

```
    void cancel(boolean mayInterrupt);
    boolean isCancelled();
    boolean isDone();
}
```

第一个get方法的调用被阻塞，直到计算完成。如果在计算完成之前，第二个方法的调用超时，抛出一个TimeoutException异常。如果运行该计算的线程被中断，两个方法都将抛出InterruptedException。如果计算已经完成，那么get方法立即返回。

如果计算还在进行，isDone方法返回false；如果完成了，则返回true。

可以用cancel方法取消该计算。如果计算还没有开始，它被取消且不再开始。如果计算处于运行之中，那么如果mayInterrupt参数为true，它就被中断。

FutureTask包装器是一种非常便利的机制，可将Callable转换成Future和Runnable，它同时实现二者的接口。例如：

```
Callable<Integer> myComputation = . . .;
FutureTask<Integer> task = new FutureTask<Integer>(myComputation);
Thread t = new Thread(task); // it's a Runnable
t.start();
. . .
Integer result = task.get(); // it's a Future
```

例14-11中的程序使用了这些概念。这个程序与前面那个寻找包含指定关键字的文件的例子相似。然而，现在我们仅仅计算匹配的文件数目。因此，我们有了一个需要长时间运行的任务，它产生一个整数值，一个Callable<Integer>的例子。

```
class MatchCounter implements Callable<Integer>
{
   public MatchCounter(File directory, String keyword) { . . . }
   public Integer call() { . . . } // returns the number of matching files
}
```

然后我们利用MatchCounter创建一个FutureTask 对象，并用来启动一个线程。

```
FutureTask<Integer> task = new FutureTask<Integer>(counter);
Thread t = new Thread(task);
t.start();
```

最后，我们打印结果。

```
System.out.println(task.get() + " matching files.");
```

当然，对get的调用会发生阻塞，直到有可获得的结果为止。

在call方法内部，使用相同的递归机制。对于每一个子目录，我们产生一个新的MatchCounter并为它启动一个线程。此外，把FutureTask对象隐藏在ArrayList<Future<Integer>>中。最后，把所有结果加起来：

```
for (Future<Integer> result : results)
   count += result.get();
```

每一次对get的调用都会发生阻塞直到结果可获得为止。当然，线程是并行运行的，因此，很可能在大致相同的时刻所有的结果都可获得。

例14-11 FutureTest.java

```
import java.io.*;
import java.util.*;
import java.util.concurrent.*;

/**
 * @version 1.0 2004-08-01
 * @author Cay Horstmann
 */
public class FutureTest
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);
      System.out.print("Enter base directory (e.g. /usr/local/jdk5.0/src): ");
      String directory = in.nextLine();
      System.out.print("Enter keyword (e.g. volatile): ");
      String keyword = in.nextLine();

      MatchCounter counter = new MatchCounter(new File(directory), keyword);
      FutureTask<Integer> task = new FutureTask<Integer>(counter);
      Thread t = new Thread(task);
      t.start();
      try
      {
         System.out.println(task.get() + " matching files.");
      }
      catch (ExecutionException e)
      {
         e.printStackTrace();
      }
      catch (InterruptedException e)
      {
      }
   }
}

/**
 * This task counts the files in a directory and its subdirectories that contain a given keyword.
 */
class MatchCounter implements Callable<Integer>
{
   /**
    * Constructs a MatchCounter.
    * @param directory the directory in which to start the search
    * @param keyword the keyword to look for
    */
   public MatchCounter(File directory, String keyword)
   {
      this.directory = directory;
      this.keyword = keyword;
   }

   public Integer call()
   {
      count = 0;
```

```
      try
      {
         File[] files = directory.listFiles();
         ArrayList<Future<Integer>> results = new ArrayList<Future<Integer>>();

         for (File file : files)
            if (file.isDirectory())
            {
               MatchCounter counter = new MatchCounter(file, keyword);
               FutureTask<Integer> task = new FutureTask<Integer>(counter);
               results.add(task);
               Thread t = new Thread(task);
               t.start();
            }
            else
            {
               if (search(file)) count++;
            }

         for (Future<Integer> result : results)
            try
            {
               count += result.get();
            }
            catch (ExecutionException e)
            {
               e.printStackTrace();
            }
      }
      catch (InterruptedException e)
      {
      }
      return count;
   }

   /**
    * Searches a file for a given keyword.
    * @param file the file to search
    * @return true if the keyword is contained in the file
    */
   public boolean search(File file)
   {
      try
      {
         Scanner in = new Scanner(new FileInputStream(file));
         boolean found = false;
         while (!found && in.hasNextLine())
         {
            String line = in.nextLine();
            if (line.contains(keyword)) found = true;
         }
         in.close();
         return found;
      }
      catch (IOException e)
      {
         return false;
```

```
        }
    }

    private File directory;
    private String keyword;
    private int count;
}
```

API java.util.concurrent.Callable<V> 5.0

- `V call()`

  运行一个将产生结果的任务。

API java.util.concurrent.Future<V> 5.0

- `V get()`
- `V get(long time, TimeUnit unit)`

  获取结果，如果没有结果可用，则阻塞直到真正得到结果超过指定的时间为止。如果不成功，第二个方法会抛出 TimeoutException异常。

- `boolean cancel(boolean mayInterrupt)`

  尝试取消这一任务的运行。如果任务已经开始，并且mayInterrupt参数值为true，它就会被中断。如果成功执行了取消操作，返回true。

- `boolean isCancelled()`

  如果任务在完成前被取消了，则返回true。

- `boolean isDone()`

  如果任务结束，无论是正常结束、中途取消或发生异常，都返回true。

API java.util.concurrent.FutureTask<V> 5.0

- `FutureTask(Callable<V> task)`
- `FutureTask(Runnable task, V result)`

  构造一个既是Future<V>又是Runnable的对象。

14.9 执行器

构建一个新的线程是有一定代价的，因为涉及与操作系统的交互。如果程序中创建了大量的生命期很短的线程，应该使用*线程池*（thread pool）。一个线程池中包含许多准备运行的空闲线程。将Runnable对象交给线程池，就会有一个线程调用run方法。当run方法退出时，线程不会死亡，而是在池中准备为下一个请求提供服务。

另一个使用线程池的理由是减少并发线程的数目。创建大量线程会大大降低性能甚至使虚拟机崩溃。如果有一个会创建许多线程的算法，应该使用一个线程数“固定的”线程池以限制并发线程的总数。

执行器（Executor）类有许多静态工厂方法用来构建线程池，表14-2中对这些方法进行

了汇总。

表14-2 执行者工厂方法

方法	描述
newCachedThreadPool	必要时创建新线程；空闲线程会被保留60秒
newFixedThreadPool	该池包含固定数量的线程；空闲线程会一直被保留
newSingleThreadExecutor	只有一个线程的“池”，该线程顺序执行每一个提交的任务（类似于Swing事件分配线程）
newScheduledThreadPool	用于预定执行而构建的固定线程池，替代java.util.Timer
newSingleThreadScheduledExecutor	用于预定执行而构建的单线程“池”

14.9.1 线程池

先来看一下表14-2中的3个方法。在第14.9.2节中，我们讨论其余的方法。newCachedThreadPool方法构建了一个线程池，对于每个任务，如果有空闲线程可用，立即让它执行任务，如果没有可用的空闲线程，则创建一个新线程。newFixedThreadPool方法构建一个具有固定大小的线程池。如果提交的任务数多于空闲的线程数，那么把得不到服务的任务放置到队列中。当其他任务完成以后再运行它们。newSingleThreadExecutor是一个退化了的大小为1的线程池：由一个线程执行提交的任务，一个接着一个。这3个方法返回实现了ExecutorService接口的ThreadPoolExecutor类的对象。

可用下面的方法之一将一个Runnable对象或Callable对象提交给ExecutorService：

```
Future<?> submit(Runnable task)
Future<T> submit(Runnable task, T result)
Future<T> submit(Callable<T> task)
```

该池会在方便的时候尽早执行提交的任务。调用submit时，会得到一个Future对象，可用来查询该任务的状态。

第一个submit方法返回一个奇怪样子的Future<?>。可以使用这样一个对象来调用isDone、cancel或isCancelled。但是，get方法在完成的时候只是简单地返回null。

第二个版本的Submit也提交一个Runnable，并且Future的get方法在完成的时候返回指定的result对象。

第三个版本的Submit提交一个Callable，并且返回的Future对象将在计算结果准备好的时候得到它。

当用完一个线程池的时候，调用shutdown。该方法启动该池的关闭序列。被关闭的执行器不再接受新的任务。当所有任务都完成以后，线程池中的线程死亡。另一种方法是调用shutdownNow。该池取消尚未开始的所有任务并试图中断正在运行的线程。

下面总结了在使用连接池时应该做的事：

1）调用Executors类中静态的方法newCachedThreadPool或newFixedThreadPool。

2）调用submit提交Runnable或Callable对象。

3）如果想要取消一个任务，或如果提交Callable对象，那就要保存好返回的Future对象。

4）当不再提交任何任务时，调用 shutdown。

例如，前面的程序例子产生了大量的生命期很短的线程，每个目录产生一个线程。例14-12中的程序使用了一个线程池来运行任务。

出于信息方面的考虑，这个程序打印出执行中池中最大的线程数。但是不能通过ExecutorService这个接口得到这一信息。因此，必须将该pool对象转型为ThreadPoolExecutor类对象。

例14-12 ThreadPoolTest.java

```
import java.io.*;
import java.util.*;
import java.util.concurrent.*;

/**
 * @version 1.0 2004-08-01
 * @author Cay Horstmann
 */
public class ThreadPoolTest
{
   public static void main(String[] args) throws Exception
   {
      Scanner in = new Scanner(System.in);
      System.out.print("Enter base directory (e.g. /usr/local/jdk5.0/src): ");
      String directory = in.nextLine();
      System.out.print("Enter keyword (e.g. volatile): ");
      String keyword = in.nextLine();

      ExecutorService pool = Executors.newCachedThreadPool();

      MatchCounter counter = new MatchCounter(new File(directory), keyword, pool);
      Future<Integer> result = pool.submit(counter);

      try
      {
         System.out.println(result.get() + " matching files.");
      }
      catch (ExecutionException e)
      {
         e.printStackTrace();
      }
      catch (InterruptedException e)
      {
      }
      pool.shutdown();

      int largestPoolSize = ((ThreadPoolExecutor) pool).getLargestPoolSize();
      System.out.println("largest pool size=" + largestPoolSize);
   }
}

/**
 * This task counts the files in a directory and its subdirectories that contain a given keyword.
 */
class MatchCounter implements Callable<Integer>
{
```

```
   /**
    * Constructs a MatchCounter.
    * @param directory the directory in which to start the search
    * @param keyword the keyword to look for
    * @param pool the thread pool for submitting subtasks
    */
   public MatchCounter(File directory, String keyword, ExecutorService pool)
   {
      this.directory = directory;
      this.keyword = keyword;
      this.pool = pool;
   }

   public Integer call()
   {
      count = 0;
      try
      {
         File[] files = directory.listFiles();
         ArrayList<Future<Integer>> results = new ArrayList<Future<Integer>>();

         for (File file : files)
            if (file.isDirectory())
            {
               MatchCounter counter = new MatchCounter(file, keyword, pool);
               Future<Integer> result = pool.submit(counter);
               results.add(result);
            }
            else
            {
               if (search(file)) count++;
            }

         for (Future<Integer> result : results)
            try
            {
               count += result.get();
            }
            catch (ExecutionException e)
            {
               e.printStackTrace();
            }
      }
      catch (InterruptedException e)
      {
      }
      return count;
   }

   /**
    * Searches a file for a given keyword.
    * @param file the file to search
    * @return true if the keyword is contained in the file
    */
   public boolean search(File file)
   {
      try
```

```
      {
         Scanner in = new Scanner(new FileInputStream(file));
         boolean found = false;
         while (!found && in.hasNextLine())
         {
            String line = in.nextLine();
            if (line.contains(keyword)) found = true;
         }
         in.close();
         return found;
      }
      catch (IOException e)
      {
         return false;
      }
   }

   private File directory;
   private String keyword;
   private ExecutorService pool;
   private int count;
}
```

API java.util.concurrent.Executors 5.0

- ExecutorService newCachedThreadPool()

  返回一个带缓存的线程池，该池在必要的时候创建线程，在线程空闲60秒之后终止线程。
- ExecutorService newFixedThreadPool(int threads)

  返回一个线程池，该池中的线程数由参数指定。
- ExecutorService newSingleThreadExecutor()

  返回一个执行器，它在一个单个的线程中依次执行各个任务。

API java.util.concurrent.ExecutorService 5.0

- Future<T> submit(Callable<T> task)
- Future<T> submit(Runnable task, T result)
- Future<?> submit(Runnable task)

  提交指定的任务去执行。
- void shutdown()

  关闭服务，会先完成已经提交的任务而不再接收新的任务。

API java.util.concurrent.ThreadPoolExecutor 5.0

- int getLargestPoolSize()

  返回线程池在该执行器生命周期中的最大尺寸。

14.9.2 预定执行

ScheduledExecutorService接口具有为预定执行（Scheduled Execution）或重复执行任务而设计的方法。它是一种允许使用线程池机制的java.util.Timer的泛化。Executors类的newScheduledThreadPool和newSingleThreadScheduledExecutor方法将返回实现了ScheduledExecutorService接口的对象。

可以预定Runnable或Callable在初始的延迟之后只运行一次。也可以预定一个Runnable对象周期性地运行。详细内容见API文档。

API `java.util.concurrent.Executors` 5.0

- `ScheduledExecutorService newScheduledThreadPool(int threads)`
  返回一个线程池，它使用给定的线程数来调度任务。
- `ScheduledExecutorService newSingleThreadScheduledExecutor()`
  返回一个执行器，它在一个单独线程中调度任务。

API `java.util.concurrent.ScheduledExecutorService` 5.0

- `ScheduledFuture<V> schedule(Callable<V> task, long time, TimeUnit unit)`
- `ScheduledFuture<?> schedule(Runnable task, long time, TimeUnit unit)`
  预定在指定的时间之后执行任务。
- `ScheduledFuture<?> scheduleAtFixedRate(Runnable task, long initialDelay, long period, TimeUnit unit)`
  预定在初始的延迟结束后，周期性地运行给定的任务，周期长度是period。
- `ScheduledFuture<?> scheduleWithFixedDelay(Runnable task, long initialDelay, long delay, TimeUnit unit)`
  预定在初始的延迟结束后周期性地给定的任务，在一次调用完成和下一次调用开始之间有长度为delay的延迟。

14.9.3 控制任务组

你已经了解了如何将一个执行器服务作为线程池使用，以提高执行任务的效率。有时，使用执行器有更有实际意义的原因，控制一组相关任务。例如，可以在执行器中使用shutdownNow方法取消所有的任务。

invokeAny方法提交所有对象到一个Callable对象的集合中，并返回某个已经完成了的任务的结果。无法知道返回的究竟是哪个任务的结果，也许是最先完成的那个任务的结果。对于搜索问题，如果你愿意接受任何一种解决方案的话，你就可以使用这个方法。例如，假定你需要对一个大整数进行因数分解计算来解码RSA密码。可以提交很多任务，每一个任务使用不同范围内的数来进行分解。只要其中一个任务得到了答案，计算就可以停止了。

invokeAll方法提交所有对象到一个Callable对象的集合中，并返回一个Future对象的列表，代表所有任务的解决方案。当计算结果可获得时，可以像下面这样对结果进行处理：

```
List<Callable<T>> tasks = . . .;
List<Future<T>> results = executor.invokeAll(tasks);
for (Future<T> result : results)
   processFurther(result.get());
```

这个方法的缺点是如果第一个任务恰巧花去了很多时间，则可能不得不进行等待。以结果按可获得的顺序保存起来更有实际意义。可以用ExecutorCompletionService来进行排列。

用常规的方法获得一个执行器。然后，构建一个ExecutorCompletionService，提交任务给完成服务（completion service）。该服务管理Future对象的阻塞队列，其中包含已经提交的任务的执行结果（当这些结果成为可用时）。这样一来，相比前面的计算，一个更有效的组织形式如下：

```
ExecutorCompletionService service = new ExecutorCompletionService(executor);
for (Callable<T> task : tasks) service.submit(task);
for (int i = 0; i < tasks.size(); i++)
   processFurther(service.take().get());
```

API java.util.concurrent.ExecutorService 5.0

- `T invokeAny(Collection<Callable<T>> tasks)`
- `T invokeAny(Collection<Callable<T>> tasks, long timeout, TimeUnit unit)`

  执行给定的任务，返回其中一个任务的结果。第二个方法若发生超时，抛出一个Timeout Exception异常。
- `List<Future<T>> invokeAll(Collection<Callable<T>> tasks)`
- `List<Future<T>> invokeAll(Collection<Callable<T>> tasks, long timeout, TimeUnit unit)`

  执行给定的任务，返回所有任务的结果。第二个方法若发生超时，抛出一个Timeout Exception异常。

API java.util.concurrent.ExecutorCompletionService 5.0

- `ExecutorCompletionService(Executor e)`

  构建一个执行器完成服务来收集给定执行器的结果。
- `Future<T> submit(Callable<T> task)`
- `Future<T> submit(Runnable task, T result)`

  提交一个任务给底层的执行器。
- `Future<T> take()`

  移除下一个已完成的结果，如果没有任何已完成的结果可用则阻塞。
- `Future<T> poll()`
- `Future<T> poll(long time, TimeUnit unit)`

  移除下一个已完成的结果，如果没有任何已完成结果可用则返回null。第二个方法将等待给定的时间。

14.10 同步器

java.util.concurrent包包含了几个能帮助人们管理相互合作的线程集的类见表14-3。这些机制具有为线程之间的*共用集结点模式*（common rendezvous patterns）提供的“预置功能”（canned functionality）。如果有一个相互合作的线程集满足这些行为模式之一，那么应该直接重用合适的库类而不要试图提供手工的锁与条件的集合。

表14-3 同步器

类	它能做什么	何时使用
CyclicBarrier	允许线程集等待直至其中预定数目的线程到达一个公共障栅（barrier），然后可以选择执行一个处理障栅的动作	当大量的线程需要在它们的结果可用之前完成时
CountDownLatch	允许线程集等待直到计数器减为0	当一个或多个线程需要等待直到指定数目的事件发生
Exchanger	允许两个线程在要交换的对象准备好时交换对象	当两个线程工作在同一数据结构的两个实例上的时候，一个向实例添加数据而另一个从实例清除数据
Semaphore	允许线程集等待直到被允许继续运行为止	限制访问资源的线程总数。如果许可数是1，常常阻塞线程直到另一个线程给出许可为止
SynchronousQueue	允许一个线程把对象交给另一个线程	在没有显式同步的情况下，当两个线程准备好将一个对象从一个线程传递到另一个时

14.10.1 信号量

概念上讲，一个信号量管理许多的*许可证*（permits）。为了通过信号量，线程通过调用acquire请求许可。许可的数目是固定的，由此限制了通过的线程数量。其他线程可以通过调用release释放许可。其实没有实际的许可对象，信号量仅维护一个计数。而且，许可不是必须由获取它的线程释放。事实上，任何线程都可以释放任意数目的许可。如果释放的许可多于可用许可的最大数目，信号量只是被设置为可用许可的最大数目。这种随意性使得信号量既具有灵活性又容易带来混乱。

信号量在1968年由Edsger Dijkstra发明，作为*同步原语*（synchronization primitive ）。Dijkstra指出信号量可以被有效地实现，并且有足够的能力解决许多常见的线程同步问题。在几乎任何一本操作系统教科书中，都能看到使用信号量实现的有界队列。当然，应用程序员不必自己实现有界队列。建议在信号量的行为能适合你面对的同步问题时才使用它，否则你会陷入思维的混乱。

一个简单的例子，一个许可数为1的信号量作为一个可由其他线程打开或关闭的门很有用。在第14.10.6节有这样的例子，工作器线程创建动画。偶尔，工作器线程等待用户按下一个按钮。工作器线程试图获得一个许可，并且它不得不等待直到按钮点击释放一个许可。

14.10.2 倒计时门栓

一个倒计时门栓（CountDownLatch）让一个线程集等待直到计数变为0。倒计时门栓是一

次性的。一旦计数为0，就不能再重用了。

一个有用的特例是计数值为1的门栓。实现一个只能通过一次的门。线程在门外等候直到另一个线程将计数器值置为0 。

举例，假定一个线程集需要一些初始的数据来完成工作。工作器线程被启动并在门外等候。另一个线程准备数据。当数据准备好的时候，调用countDown，所有工作器线程就可以继续运行了。

然后，可以使用第二个门栓检查什么时候所有工作器线程完成工作。用线程数初始化门栓。每个工作器线程在结束前将门栓计数减1。另一个获取工作结果的线程在门外等待，一旦所有工作器线程终止该线程继续运行。

14.10.3 障栅

CyclicBarrier类实现了一个*集结点*（rendezvous）称为*障栅*（barrier）。考虑大量线程运行在一次计算的不同部分的情形。当所有部分都准备好时，需要把结果组合在一起。当一个线程完成了它的那部分任务后，我们让它运行到障栅处。一旦所有的线程都到达了这个障栅，障栅就撤销，线程就可以继续运行。

下面是其细节。首先，构造一个障栅，并给出参与的线程数：

```
CyclicBarrier barrier = new CyclicBarrier(nthreads);
```

每一个线程做一些工作，完成后在障栅上调用await ：

```
public void run()
{
   doWork();
   barrier.await();
   . . .
}
```

await方法有一个可选的超时参数：

```
barrier.await(100, TimeUnit.MILLISECONDS);
```

如果任何一个在障栅上等待的线程离开了障栅，那么障栅就被破坏了（线程可能离开是因为它调用await时设置了超时，或者因为它被中断了）。在这种情况下，所有其他线程的await方法抛出BrokenBarrierException异常。那些已经在等待的线程立即终止await的调用。

可以提供一个可选的*障栅动作*（barrier action），当所有线程到达障栅的时候就会执行这一动作。

```
Runnable barrierAction = . . .;
CyclicBarrier barrier = new CyclicBarrier(nthreads, barrierAction);
```

该动作可以收集那些单个线程的运行结果。

障栅被称为是*循环的*（cyclic），因为可以在所有等待线程被释放后被重用。在这一点上，有别于CountDownLatch，CountDownLatch只能被使用一次。

14.10.4 交换器

当两个线程在同一个数据缓冲区的两个实例上工作的时候，就可以使用*交换器*(Exchanger)。

典型的情况是，一个线程向缓冲区填入数据，另一个线程消耗这些数据。当它们都完成以后，相互交换缓冲区。

14.10.5 同步队列

同步队列是一种将生产者与消费者线程配对的机制。当一个线程调用SynchronousQueue 的put方法时，它会阻塞直到另一个线程调用take方法为止，反之亦然。与Exchanger的情况不同，数据仅仅沿一个方向传递，从生产者到消费者。

即使SynchronousQueue类实现了BlockingQueue接口，概念上讲，它依然不是一个队列。它没有包含任何元素，它的size方法总是返回0。

14.10.6 例子：暂停动画与恢复动画

考虑一个要做某项工作的程序，更新屏幕的显示，在等待用户查看结果之后按下按钮继续，然后完成工作的下一步。

许可计数为1的信号量可以用来处理工作器线程和事件分配线程之间的同步。工作器线程在准备好要暂停的时候调用acquire。只要用户点击Continue按钮，GUI线程就调用release。

如果在工作器线程准备好的时候，用户多次点击该按钮会发生什么呢？毕竟因为只有一个许可证可用，许可计数为1。

例14-13中的程序使用这种思路工作。程序以动画形式展示排序算法。工作器线程对数组进行排序，周期性地停止，等待用户继续给出许可。用户看到算法当前状态的绘图会感到满意，并按下Continue按钮允许工作器线程进行下一步处理。

这里不打算让读者为排序算法烦恼，于是，调用Arrays.sort，它实现归并算法。要暂停该算法，我们提供了一个等待信号量的Comparator对象。因此，每当算法比较两个元素的时候，暂停动画。绘制数组的当前值并加亮显示被比较的元素（见图14-8）。

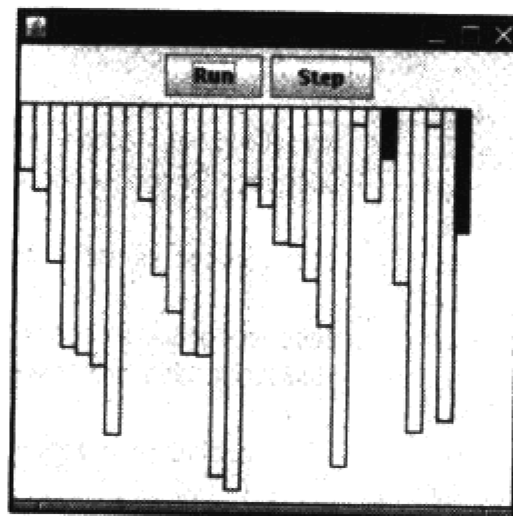


图14-8 动画演示排序算法

> **注释**：动画展示了较小的有序段归并到较大的有序段，但是它并不那么精确。mergesort算法使用第二个数组存放我们无法看到的临时值。这个例子的要点是不要过于用心地钻研排序算法，而是展示如何使用信号量用来暂停工作器线程。

例14-13 AlgorithmAnimation.java

```
import java.awt.*;
import java.awt.geom.*;
import java.awt.event.*;
import java.util.*;
import java.util.concurrent.*;
import javax.swing.*;

/**
 * This program animates a sort algorithm.
 * @version 1.01 2007-05-18
```

```
 * @author Cay Horstmann
 */
public class AlgorithmAnimation
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new AnimationFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
 * This frame shows the array as it is sorted, together with buttons to single-step the
 * animation or to run it without interruption.
 */
class AnimationFrame extends JFrame
{
   public AnimationFrame()
   {
      ArrayComponent comp = new ArrayComponent();
      add(comp, BorderLayout.CENTER);

      final Sorter sorter = new Sorter(comp);

      JButton runButton = new JButton("Run");
      runButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               sorter.setRun();
            }
         });

      JButton stepButton = new JButton("Step");
      stepButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               sorter.setStep();
            }
         });

      JPanel buttons = new JPanel();
      buttons.add(runButton);
      buttons.add(stepButton);
      add(buttons, BorderLayout.NORTH);
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      Thread t = new Thread(sorter);
      t.start();
```

```
   }

   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 300;
}

/**
 * This runnable executes a sort algorithm. When two elements are compared, the algorithm
 * pauses and updates a component.
 */
class Sorter implements Runnable
{
   /**
    * Constructs a Sorter.
    * @param values the array to be sorted
    * @param comp the component on which to display the sorting progress
    */
   public Sorter(ArrayComponent comp)
   {
      values = new Double[VALUES_LENGTH];
      for (int i = 0; i < values.length; i++)
         values[i] = new Double(Math.random());
      this.component = comp;
      this.gate = new Semaphore(1);
      this.run = false;
   }

   /**
    * Sets the sorter to "run" mode. Called on the event dispatch thread.
    */
   public void setRun()
   {
      run = true;
      gate.release();
   }

   /**
    * Sets the sorter to "step" mode. Called on the event dispatch thread.
    */
   public void setStep()
   {
      run = false;
      gate.release();
   }

   public void run()
   {
      Comparator<Double> comp = new Comparator<Double>()
         {
            public int compare(Double i1, Double i2)
            {
               component.setValues(values, i1, i2);
               try
               {
                  if (run) Thread.sleep(DELAY);
                  else gate.acquire();
               }
```

```
                catch (InterruptedException exception)
                {
                   Thread.currentThread().interrupt();
                }
                return i1.compareTo(i2);
             }
          };
       Arrays.sort(values, comp);
       component.setValues(values, null, null);
    }

    private Double[] values;
    private ArrayComponent component;
    private Semaphore gate;
    private static final int DELAY = 100;
    private volatile boolean run;
    private static final int VALUES_LENGTH = 30;
}

/**
 * This component draws an array and marks two elements in the array.
 */
class ArrayComponent extends JComponent
{
   /**
    * Sets the values to be painted. Called on the sorter thread.
    * @param values the array of values to display
    * @param marked1 the first marked element
    * @param marked2 the second marked element
    */
   public synchronized void setValues(Double[] values, Double marked1, Double marked2)
   {
      this.values = values.clone();
      this.marked1 = marked1;
      this.marked2 = marked2;
      repaint();
   }

   public synchronized void paintComponent(Graphics g) // Called on the event dispatch thread
   {
      if (values == null) return;
      Graphics2D g2 = (Graphics2D) g;
      int width = getWidth() / values.length;
      for (int i = 0; i < values.length; i++)
      {
         double height = values[i] * getHeight();
         Rectangle2D bar = new Rectangle2D.Double(width * i, 0, width, height);
         if (values[i] == marked1 || values[i] == marked2) g2.fill(bar);
         else g2.draw(bar);
      }
   }

   private Double marked1;
   private Double marked2;
   private Double[] values;
}
```

java.util.concurrent.CyclicBarrier 5.0

- `CyclicBarrier(int parties)`
- `CyclicBarrier(int parties, Runnable barrierAction)`

  构建一个线程数目是parties的循环障栅。当所有的线程都在障栅上调用await之后，执行barrierAction。
- `int await()`
- `int await(long time, TimeUnit unit)`

  等待直到所有的线程在障栅上调用await或者时间超时为止，在这种情况下会抛出TimeoutException异常。成功时，返回这个线程的序号。第一个线程的序号为parties－1，最后一个线程是0。

java.util.concurrent.CountDownLatch 5.0

- `CountdownLatch(int count)`

  用给定的计数构建一个倒计时门栓。
- `void await()`

  等待这个门栓的计数降为0 。
- `boolean await(long time, TimeUnit unit)`

  等待这个门栓的计数降为0或者时间超时。如果计数为0返回true，如果超时返回false。
- `public void countDown()`

  递减这个门栓的计数值。

java.util.concurrent.Exchanger<V> 5.0

- `V exchange(V item)`
- `V exchange(V item, long time, TimeUnit unit)`

  阻塞直到另一个线程调用这个方法，然后，同其他线程交换item，并返回其他线程的item。第二个方法时间超时时，抛出TimeoutException异常。

java.util.concurrent.SynchronousQueue<V> 5.0

- `SynchronousQueue()`
- `SynchronousQueue(boolean fair)`

  构建一个允许线程提交item的同步队列。如果fair为true，队列优先照顾等待了最长时间的线程。
- `void put(V item)`

  阻塞直到另一个线程调用take来获取item。
- `V take()`

  阻塞直到另一个线程调用put。返回另一个线程提供的item。

API java.util.concurrent.Semaphore 5.0

- `Semaphore(int permits)`
- `Semaphore(int permits, boolean fair)`

  用给定的许可数目为最大值构建一个信号量。如果 fair为true，队列优先照顾等待了最长时间的线程。
- `void acquire()`

  等待获得一个许可。
- `boolean tryAcquire()`

  尝试获得一个许可，如果没有许可是可用的，返回false。
- `boolean tryAcquire(long time, TimeUnit unit)`

  尝试在给定时间内获得一个许可，如果没有许可是可用的，返回false。
- `void release()`

  释放一个许可。

14.11 线程与Swing

在有关本章的介绍里已经提到，在程序中使用线程的理由之一是提高程序的响应性能。当程序需要做某些耗时的工作时，应该启动另一个工作器线程而不是阻塞用户接口。

但是，必须认真考虑工作器线程在做什么，因为这或许令人惊讶，Swing不是线程安全的。如果你试图在多个线程中操纵用户界面的元素，那么用户界面可能崩溃。

要了解这一问题，运行例14-14的测试程序。当你点击Bad按钮时，一个新的线程将启动，它的run方法操作一个组合框，随机地添加值和删除值。

```
public void run()
{
   try
   {
      while (true)
      {
         int i = Math.abs(generator.nextInt());
         if (i % 2 == 0)
            combo.insertItemAt(new Integer(i), 0);
         else if (combo.getItemCount() > 0)
            combo.removeItemAt(i % combo.getItemCount());
         sleep(1);
      }
      catch (InterruptedException e) {}
   }
}
```

试试看。点击Bad按钮。点击几次组合框，移动滚动条，移动窗口，再次点击Bad按钮，不断点击组合框。最终，你会看到一个异常报告（见图14-9）。

发生了什么？当把一个元素插入组合框时，组合框将产生一个事件来更新显示。然后，显示代码开始运行，读取组合框的当前大小并准备显示这个值。但是，工作器线程保持运行，有

时候会造成组合框中值的数目减少。显示代码认为组合框中的值比实际的数量多，于是会访问不存在的值，触发ArrayIndexOutOfBounds异常。

在显示时对组合框加锁可以避免这种情况出现。但是，Swing的设计者决定不再付出更多的努力实现Swing线程安全，有两个原因。首先，同步需要时间，而且，已经没有人想要降低Swing的速度。更重要的是，Swing小组调查了其他小组在线程安全的用户界面工具包方面的经验。他们的发现并不令人鼓舞。使用线程安全包的程序员被同步命令搞昏了头，常常编写出容易造成死锁的程序。

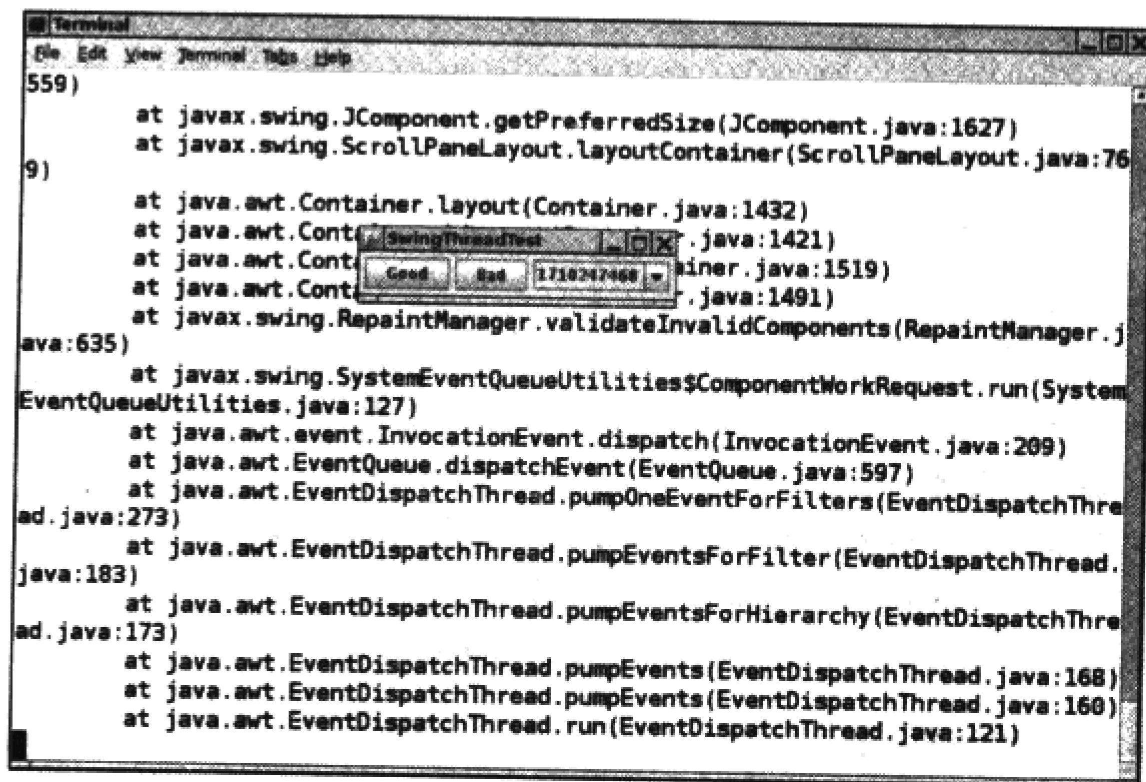


图14-9 控制台的异常报告

14.11.1 运行耗时的任务

将线程与Swing一起使用时，必须遵循两个简单的原则。

- 如果一个动作需要花费很长时间，在一个独立的工作器线程中做这件事不要在事件分配线程中做。
- 除了事件分配线程，不要在任何线程中接触Swing组件。

制定第一条规则的理由易于理解。如果花很多时间在事件分配线程上，应用程序像“死了”一样，因为它不响应任何事件。特别是，事件分配线程应该永远不要进行input/output调用，这有可能会阻塞，并且应该永远不要调用sleep。（如果需要等待指定的时间，使用定时器事件。）

第二条规则在Swing编程中通常称为*单一线程规则*（single-thread rule）。我们在后面的内容中进一步讨论。

这两条规则看起来彼此冲突。假定要启动一个独立的线程运行一个耗时的任务。线程工作的时候，通常要更新用户界面中指示执行的进度。任务完成的时候，要再一次更新GUI界面。但是，不能从自己的线程接触Swing组件。例如，如果要更新进度条或标签文本，不能从线程中设置它的值。

要解决这一问题，在任何线程中，可以使用两种有效的方法向事件队列添加任意的动作。例如，假定想在一个线程中周期性地更新标签来表明进度。不可以从自己的线程中调用label.setText，而应该使用EventQueue类的invokeLater方法和invokeAndWait方法使所调用的方法在事件分配线程中执行。

应该将Swing代码放置到实现Runnable接口的类的run方法中。然后，创建该类的一个对象，将其传递给静态的invokeLater或invokeAndWait方法。例如，下面是如何更新标签内容的代码：

```
EventQueue.invokeLater(new
   Runnable()
   {
      public void run()
      {
         label.setText(percentage + "% complete");
      }
   });
```

当事件放入事件队列时，invokeLater方法立即返回，而run方法被异步执行。invokeAndWait方法等待直到run方法确被实执行过为止。

在更新进度标签时，invokeLater方法更适宜。用户更希望让工作器线程有更快完成工作而不是得到更加精确的进度指示器。

这两种方法都是在事件分配线程中执行run方法。没有新的线程被创建。

例14-14演示了如何使用invokeLater方法安全地修改组合框的内容。如果点击Good按钮，线程插入或移除数字。但是，实际的修改是发生在事件分配线程中。

例14-14 SwingThreadTest.java

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;

/**
 * This program demonstrates that a thread that runs in parallel with the event dispatch thread
 * can cause errors in Swing components.
 * @version 1.23 2007-05-17
 * @author Cay Horstmann
 */
public class SwingThreadTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               SwingThreadFrame frame = new SwingThreadFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
```

```
}

/**
 * This frame has two buttons to fill a combo box from a separate thread. The "Good" button
 * uses the event queue; the "Bad" button modifies the combo box directly.
 */
class SwingThreadFrame extends JFrame
{
   public SwingThreadFrame()
   {
      setTitle("SwingThreadTest");

      final JComboBox combo = new JComboBox();
      combo.insertItemAt(Integer.MAX_VALUE, 0);
      combo.setPrototypeDisplayValue(combo.getItemAt(0));
      combo.setSelectedIndex(0);

      JPanel panel = new JPanel();

      JButton goodButton = new JButton("Good");
      goodButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               new Thread(new GoodWorkerRunnable(combo)).start();
            }
         });
      panel.add(goodButton);
      JButton badButton = new JButton("Bad");
      badButton.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               new Thread(new BadWorkerRunnable(combo)).start();
            }
         });
      panel.add(badButton);

      panel.add(combo);
      add(panel);
      pack();
   }
}

/**
 * This runnable modifies a combo box by randomly adding and removing numbers. This can result
 * in errors because the combo box methods are not synchronized and both the worker thread
 * and the event dispatch thread access the combo box.
 */
class BadWorkerRunnable implements Runnable
{
   public BadWorkerRunnable(JComboBox aCombo)
   {
      combo = aCombo;
      generator = new Random();
   }

```

```
   public void run()
   {
      try
      {
         while (true)
         {
            int i = Math.abs(generator.nextInt());
            if (i % 2 == 0) combo.insertItemAt(i, 0);
            else if (combo.getItemCount() > 0) combo.removeItemAt(i % combo.getItemCount());
            Thread.sleep(1);
         }
      }
      catch (InterruptedException e)
      {
      }
   }

   private JComboBox combo;
   private Random generator;
}

/**
 * This runnable modifies a combo box by randomly adding and removing numbers. In order to
 * ensure that the combo box is not corrupted, the editing operations are forwarded to the
 * event dispatch thread.
 */
class GoodWorkerRunnable implements Runnable
{
   public GoodWorkerRunnable(JComboBox aCombo)
   {
      combo = aCombo;
      generator = new Random();
   }

   public void run()
   {
      try
      {
         while (true)
         {
            EventQueue.invokeLater(new Runnable()
               {
                  public void run()
                  {
                     int i = Math.abs(generator.nextInt());
                     if (i % 2 == 0) combo.insertItemAt(i, 0);
                     else if (combo.getItemCount() > 0) combo.removeItemAt(i
                           % combo.getItemCount());
                  }
               });
            Thread.sleep(1);
         }
      }
      catch (InterruptedException e)
      {
      }
   }
```

```
    private JComboBox combo;
    private Random generator;
}
```

API java.awt.EventQueue 1.1

- `static void invokeLater(Runnable runnable) 1.2`

  在待处理的线程被处理之后，让runnable对象的run方法在事件分配线程中执行。

- `static void invokeAndWait(Runnable runnable) 1.2`

  在待处理的线程被处理之后，让runnable对象的run方法在事件分配线程中执行。该调用会阻塞，直到run方法终止。

- `static boolean isDispatchThread() 1.2`

  如果执行这一方法的线程是事件分配线程，返回true。

14.11.2 使用Swing工作器

当用户发布一条处理过程很耗时的命令时，你可能打算启动一个新的线程来完成这个工作。如同上一节介绍的那样，线程应该使用EventQueue.invokeLater方法来更新用户界面。

有人已经创建了很方便的类，让人们轻松地完成这样的任务，并且，这些类中的一个已经编入Java SE 6。本节我们介绍SwingWorker类。

例14-15中的程序有加载文本文件的命令和取消加载过程的命令。应该用一个长的文件来测试这个程序，例如The Count of Monte Cristo的全文，它在本书的附赠代码的gutenberg目录下。该文件在一个单独的线程中加载。在读取文件的过程中，Open菜单项被禁用，Cancel菜单项为可用（见图14-10）。读取每一行后，状态条中的线性计数器被更新。读取过程完成之后，Open菜单项重新变为可用，Cancel项被禁用，状态行文本置为Done。

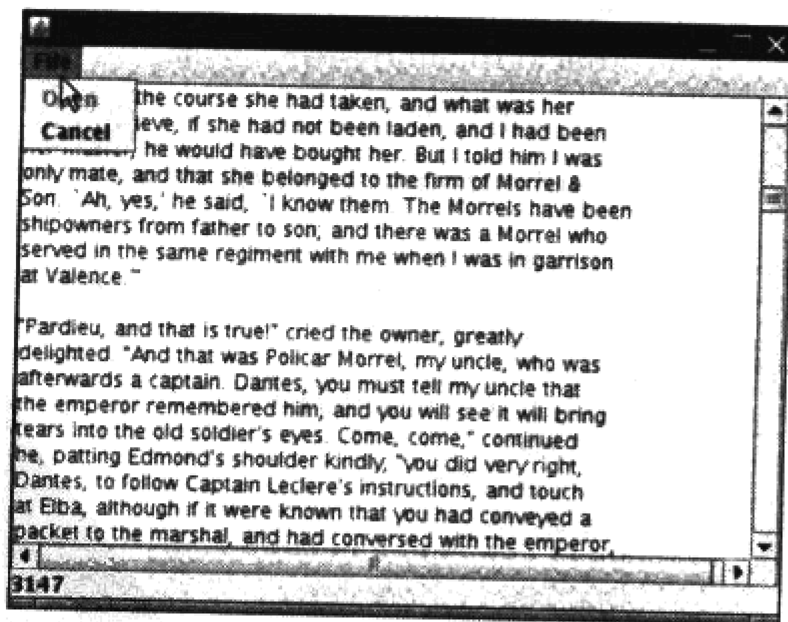


图14-10 在独立线程中加载文件

这个例子展示了后台任务的典型UI活动：

- 在每一个工作单位完成之后，更新UI来显示进度。
- 整个工作完成之后，对UI做最后的更新。

SwingWorker类使得实现这一任务轻而易举。覆盖doInBackground方法来完成耗时的工作，不时地调用publish来报告工作进度。这一方法在工作器线程中执行。publish方法使得process方法在事件分配线程中执行来处理进度数据。当工作完成时，done方法在事件分配线程中被调用以便完成UI的更新。

每当要在工作器线程中做一些工作时，构建一个新的工作器（每一个工作器对象只能被使用一次）。然后调用execute方法。典型的方式是在事件分配线程中调用execute，但没有这样的需求。

假定工作器产生某种类型的结果；因此，SwingWorker<T, V>实现Future<T>。这一结果可以通过Future接口的get方法获得。由于get方法阻塞直到结果成为可用，因此不要在调用execute之后马上调用它。只在已经知道工作完成时调用它，是最为明智的。典型地，可以从done方法调用get。（有时，没有调用get的需求，处理进度数据就是你所需要的。）

中间的进度数据以及最终的结果可以是任何类型。SwingWorker类有3种类型作为类型参数。SwingWorker<T, V>产生类型为T的结果以及类型为V的进度数据。

要取消正在进行的工作，使用Future接口的cancel方法。当该工作被取消的时候，get方法抛出CancellationException异常。

正如前面已经提到的，工作器线程对publish的调用会导致在事件分配线程上的process的调用。为了提高效率，几个对publish的调用结果，可用对process的一次调用成批处理。process方法接收一个包含所有中间结果的列表<V>。

把这一机制用于读取文本文件的工作中。正如所看到的，JTextArea相当慢。在一个长的文本文件（比如，The Count of Monte Cristo）中追加行会花费相当可观的时间。

为了向用户展示进度，要在状态行中显示读入的行数。因此，进度数据包含当前行号以及文本的当前行。将它们打包到一个普通的内部类中：

```
private class ProgressData
{
   public int number;
   public String line;
}
```

最后的结果是已经读入StringBuilder的文本。因此，需要一个SwingWorker<StringBuilder, ProgressData>。

在doInBackground方法中，读取一个文件，每次一行。在读取每一行之后，调用publish方法发布行号和当前行的文本。

```
@Override public StringBuilder doInBackground() throws IOException, InterruptedException
{
   int lineNumber = 0;
   Scanner in = new Scanner(new FileInputStream(file));
   while (in.hasNextLine())
   {
      String line = in.nextLine();
      lineNumber++;
      text.append(line);
      text.append("\n");
      ProgressData data = new ProgressData();
      data.number = lineNumber;
      data.line = line;
      publish(data);
      Thread.sleep(1); // to test cancellation; no need to do this in your programs
   }
   return text;
}
```

在读取每一行之后休眠1毫秒，以便不使用重读就可以检测取消动作，但是，不要使用休眠来减慢程序的执行速度。如果对这一行加注解，会发现The Count of Monte Cristo 的加载相

当快，只有几批用户接口更新。

✔ **注释**：从工作器线程来更新文本区可以使这个程序的处理相当顺畅，但是，对大多数Swing组件来说不可能做到这一点。这里，给出一种通用的方法，其中所有组件的更新都出现在事件分配线程中。

在这个process方法中，忽略除最后一行行号之外的所有行号，然后，我们把所有的行拼接在一起用于文本区的一次更新。

```
@Override public void process(List<ProgressData> data)
{
   if (isCancelled()) return;
   StringBuilder b = new StringBuilder();
   statusLine.setText("" + data.get(data.size() - 1).number);
   for (ProgressData d : data) { b.append(d.line); b.append("\n"); }
   textArea.append(b.toString());
}
```

在done方法中，文本区被更新为完整的文本，并且Cancel菜单项被禁用。

在Open菜单项的事件监听器中，工作器是如何启动的。这一简单的技术允许人们在保持对用户界面的正常响应的同时，执行耗时的任务。

例14-15 SwingWorkerTest.java

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import java.util.List;
import java.util.concurrent.*;

import javax.swing.*;

/**
 * This program demonstrates a worker thread that runs a potentially time-consuming task.
 * @version 1.1 2007-05-18
 * @author Cay Horstmann
 */
public class SwingWorkerTest
{
   public static void main(String[] args) throws Exception
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               JFrame frame = new SwingWorkerFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

```

```
/**
 * This frame has a text area to show the contents of a text file, a menu to open a file and
 * cancel the opening process, and a status line to show the file loading progress.
 */
class SwingWorkerFrame extends JFrame
{
   public SwingWorkerFrame()
   {
      chooser = new JFileChooser();
      chooser.setCurrentDirectory(new File("."));

      textArea = new JTextArea();
      add(new JScrollPane(textArea));
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      statusLine = new JLabel(" ");
      add(statusLine, BorderLayout.SOUTH);

      JMenuBar menuBar = new JMenuBar();
      setJMenuBar(menuBar);

      JMenu menu = new JMenu("File");
      menuBar.add(menu);

      openItem = new JMenuItem("Open");
      menu.add(openItem);
      openItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               // show file chooser dialog
               int result = chooser.showOpenDialog(null);

               // if file selected, set it as icon of the label
               if (result == JFileChooser.APPROVE_OPTION)
               {
                  textArea.setText("");
                  openItem.setEnabled(false);
                  textReader = new TextReader(chooser.getSelectedFile());
                  textReader.execute();
                  cancelItem.setEnabled(true);
               }
            }
         });

      cancelItem = new JMenuItem("Cancel");
      menu.add(cancelItem);
      cancelItem.setEnabled(false);
      cancelItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               textReader.cancel(true);
            }
         });
   }
```

```java

   private class ProgressData
   {
      public int number;
      public String line;
   }

   private class TextReader extends SwingWorker<StringBuilder, ProgressData>
   {
      public TextReader(File file)
      {
         this.file = file;
      }

      // the following method executes in the worker thread; it doesn't touch Swing components

      @Override
      public StringBuilder doInBackground() throws IOException, InterruptedException
      {
         int lineNumber = 0;
         Scanner in = new Scanner(new FileInputStream(file));
         while (in.hasNextLine())
         {
            String line = in.nextLine();
            lineNumber++;
            text.append(line);
            text.append("\n");
            ProgressData data = new ProgressData();
            data.number = lineNumber;
            data.line = line;
            publish(data);
            Thread.sleep(1); // to test cancellation; no need to do this in your programs
         }
         return text;
      }

      // the following methods execute in the event dispatch thread

      @Override
      public void process(List<ProgressData> data)
      {
         if (isCancelled()) return;
         StringBuilder b = new StringBuilder();
         statusLine.setText("" + data.get(data.size() - 1).number);
         for (ProgressData d : data)
         {
            b.append(d.line);
            b.append("\n");
         }
         textArea.append(b.toString());
      }

      @Override
      public void done()
      {
         try
         {
```

```
               StringBuilder result = get();
               textArea.setText(result.toString());
               statusLine.setText("Done");
            }
            catch (InterruptedException ex)
            {
            }
            catch (CancellationException ex)
            {
               textArea.setText("");
               statusLine.setText("Cancelled");
            }
            catch (ExecutionException ex)
            {
               statusLine.setText("" + ex.getCause());
            }

            cancelItem.setEnabled(false);
            openItem.setEnabled(true);
         }

         private File file;
         private StringBuilder text = new StringBuilder();
      };

   private JFileChooser chooser;
   private JTextArea textArea;
   private JLabel statusLine;
   private JMenuItem openItem;
   private JMenuItem cancelItem;
   private SwingWorker<StringBuilder, ProgressData> textReader;

   public static final int DEFAULT_WIDTH = 450;
   public static final int DEFAULT_HEIGHT = 350;
}
```

API javax.swing.SwingWorker<T, V> 6

- abstract T doInBackground()

  覆盖这一方法来执行后台的任务并返回这一工作的结果。
- void process(List<V> data)

  覆盖这一方法来处理事件分配线程中的中间进度数据。
- void publish(V... data)

  传递中间进度数据到事件分配线程。从doInBackground调用这一方法。
- void execute()

  为工作器线程的执行预定这个工作器。
- SwingWorker.StateValue getState()

  得到这个工作器线程的状态，值为PENDING、STARTED或 DONE之一。

14.11.3 单一线程规则

每一个Java应用程序都开始于主线程中的main方法。在Swing程序中，main方法的生命期是很短的。它在事件分配线程中规划用户界面的构造然后退出。在用户界面构造之后，事件分配线程会处理事件通知，例如调用actionPerformed或paintComponent。其他线程在后台运行，例如将事件放入事件队列的进程，但是那些线程对应用程序员是不可见的。

本章前面介绍了单一线程规则："除了事件分配线程，不要在任何线程中接触Swing组件。"本节进一步研究此规则。

对于单一线程规则存在一些例外情况。

- 可在任一个线程里添加或移除事件监听器。当然该监听器的方法会在事件分配线程中被触发。
- 只有很少的Swing方法是线程安全的。在API文档中用这样的句子特别标明："*尽管大多数Swing方法不是线程安全的，但这一方法是。*"在这些线程安全的方法中最有用的是：

```
JTextComponent.setText
JTextArea.insert
JTextArea.append
JTextArea.replaceRange
JComponent.repaint
JComponent.revalidate
```

注释：*在本书中多次使用repaint方法，但是，revalidate方法不怎么常见。这样做的目的是在内容改变之后强制执行组件布局。传统的AWT有一个validate方法强制执行组件布局。对于Swing组件，应该调用revalidate方法。（但是，要强制执行JFrame的布局，仍然要调用validate方法，因为JFrame是一个Component不是一个JComponent。）*

历史上，单一线程规则是更加随意的。任何线程都可以构建组件，设置优先级，将它们添加到容器中，只要这些组件没有一个是已经被实现的（realized）。如果组件可以接收paint事件或validation事件，组件被实现。一旦调用组件的setVisible(true)或pack(!)方法或者组件已经被添加到已经被实现的容器中，就出现这样的情况。

单一线程规则的这一版本是便利的，它允许在main方法中创建GUI，然后，在应用程序的顶层框架调用setVisible(true)。在事件分配线程上没有令人讨厌的Runnable的安排。

遗憾的是，一些组件的实现者没有注意原来的单一线程规则的微妙之处。他们在事件分配线程启动活动，而没有检查组件是否是被实现的。例如，如果在JTextComponent上调用setSelectionStart或setSelectionEnd，在事件分配线程中安排了一个插入符号的移动，即使该组件不是可见的。

检测并定位这些问题可能会好些，但是Swing的设计者没有走这条轻松的路。他们认定除了使用事件分配线程之外，从任何其他线程访问组件永远都是不安全的。因此，你需要在事件分配线程构建用户界面，像程序示例中那样调用EventQueue.invokeLater。

当然，有不少程序使用旧版的单一线程规则，在主线程初始化用户界面。那些程序有一定

的风险，某些用户界面的初始化会引起事件分配线程的动作与主线程的动作发生冲突。如同我们在第7章讲到的，不要让自己成为少数不幸的人之一，为时有时无的线程bug烦恼并花费时间。因此，一定要遵循严谨的单一线程规则。

现在读者已经读到《Java核心技术 卷I》的末尾。这一卷涵盖了Java程序设计语言的基础知识以及大多数编程项目所需要的标准库中的部分内容。希望读者在学习Java基础知识的过程中感到愉快并得到了有用的信息。有关高级知识内容，如网络、高级的AWT/Swing、安全性以及国际化，请阅读卷II。